Roadmap for the Development of a Linear Algebra Library for Exascale Computing

SLATE: Software for Linear Algebra Targeting Exascale

Ahmad Abdelfattah
Hartwig Anzt
Aurelien Bouteiller
Anthony Danalis
Jack Dongarra
Mark Gates
Azzam Haidar
Jakub Kurzak
Piotr Luszczek
Stanimire Tomov
Stephen Wood
Panruo Wu
Ichitaro Yamazaki
Asim YarKhan

Innovative Computing Laboratory, University of Tennessee

July 1, 2017

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

The authors would like to thank Shane Story, Konstantin Arthurov, and Murat Guney, from the Intel MKL team, for tharing their insights about the inner workings of MKL.

The authors would like to thank Clint Whaley, the author of ATLAS, for sharing his insights about the inner workings of ATLAS.

Revision	Notes
06-2017	first publication

```
@techreport{abdelfattah2017roadmap,
  author={Abdelfattah, Ahmad and Anzt, Hartwig and Bouteiller, Aurelien and
          Danalis, Anthony and Dongarra, Jack and Gates, Mark and
          Haidar, Azzam and Kurzak, Jakub and Luszczek, Piotr and
          Tomov, Stanimire and Wood, Stephen and Wu, Panruo and
          Yamazaki, Ichitaro and YarKhan, Asim},
  title={Roadmap for the Development of a Linear Algebra Library
         for Exascale Computing: {SLATE}: Software for Linear Algebra
         Targeting Exascale},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2017},
  month={June},
  type={SLATE Working Note},
  number={1},
  note={revision 06-2017}
}
```

Contents

List of Figures

List of Tables

CHAPTER 1

Preface

ScaLAPACK was first released in 1995, 22 years ago. To put it in perspective, this was one year after version 1.0 of the MPI standard was released, and two years before the OpenMP Fortran 1.0 specification was released. The fastest machine on the TOP500 list was the Japanese Numerical Wind Tunnel, with peak performance of 235.8 GFLOPS. This was the year when Microsoft acquired NCSA Mosaic to build the Internet Explorer.

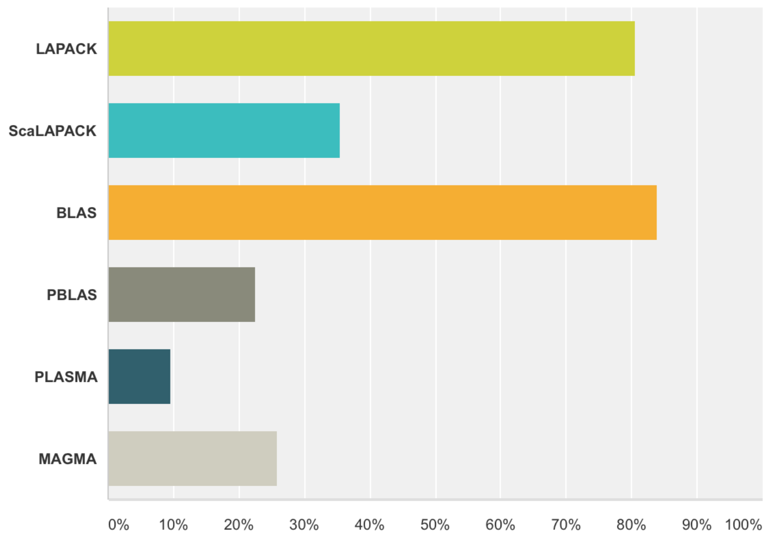
The past two decades witnessed tectonic shifts in the hardware technology, followed by paradigm shifts in the software technology, and a plethora of algorithmic innovation in scientific computing. At the same time, no viable replacement for ScaLAPACK emerged, that would channel this technological progress into a robust software package. SLATE is meant to be this replacement, and the objective of this document is to provide an overview of the cutting-edge solutions required to accomplish that mission.

CHAPTER 2

ECP Applications Survey

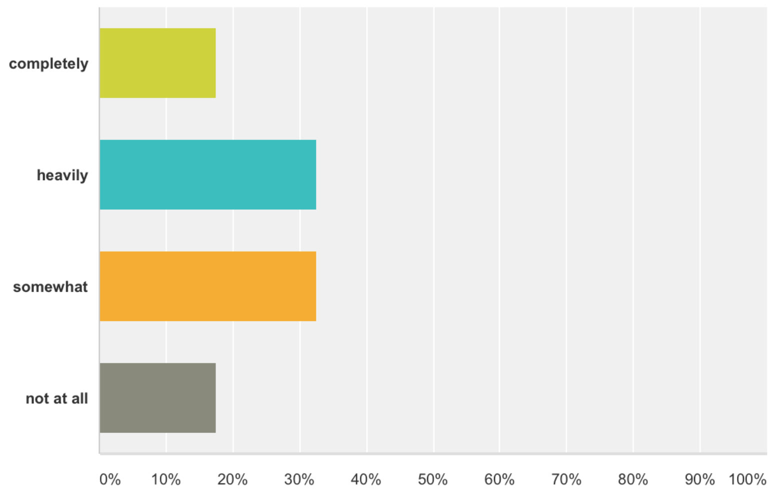
In February 2017, the SLATE team circulated a survey to the ECP applications teams, to asses their needs for dense linear algebra functionality. 40 responses were collected, 25 from project's PIs and co-PIs and 15 from other team members. Here, the responses to the most important questions are summarized.

2.1 Results



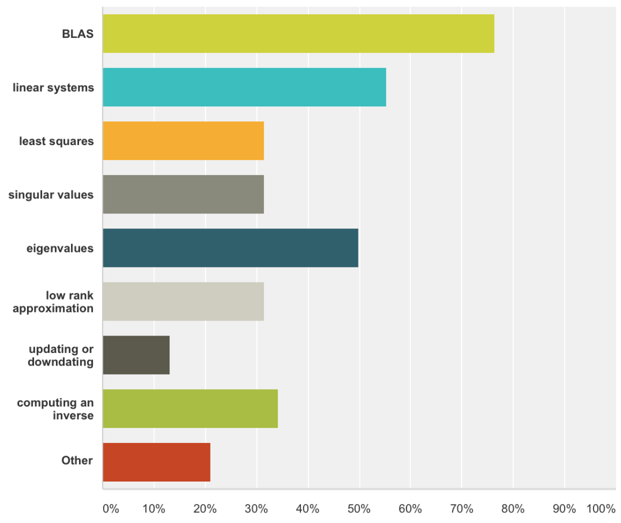
LAPACK	25
ScaLAPACK	11
BLAS	26
PBLAS	7
PLASMA	3
MAGMA	8

Figure 2.1: Is your application directly calling any of the following packages? Mark all that apply.



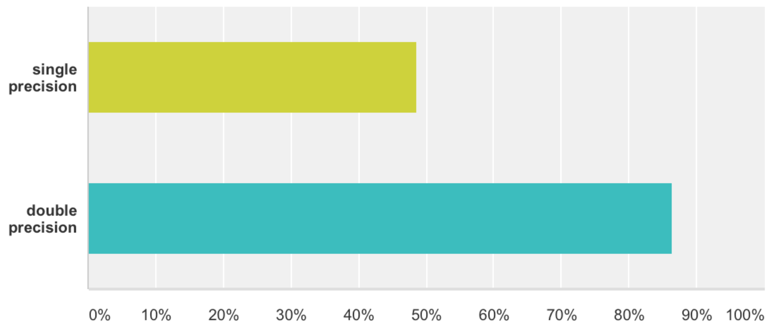
completely	7
heavily	13
somewhat	13
not at all	7

Figure 2.2: To what extent does your application rely on dense or band linear algebra operations?



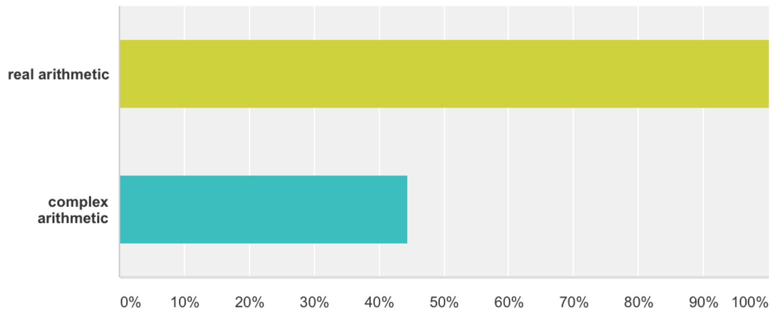
BLAS	29
linear systems	21
least squares	12
singular values	12
eigenvalues	19
low rank approximation	12
updating or downdating	5
computing an inverse	13
Other	8

Figure 2.3: What linear algebra routines are you using? Mark all that apply.



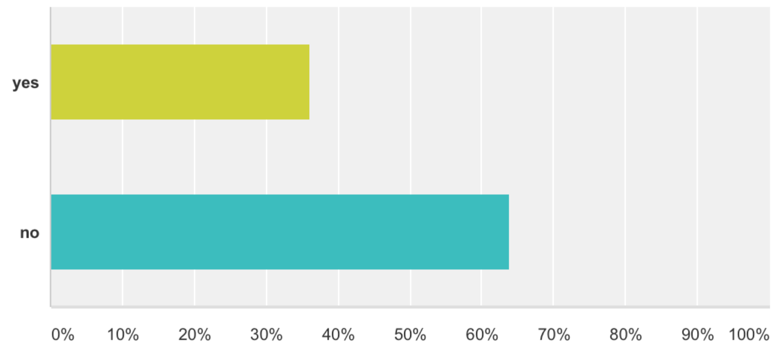
single	18
double	32

Figure 2.4: Which precisions are you using? Mark both if applicable.



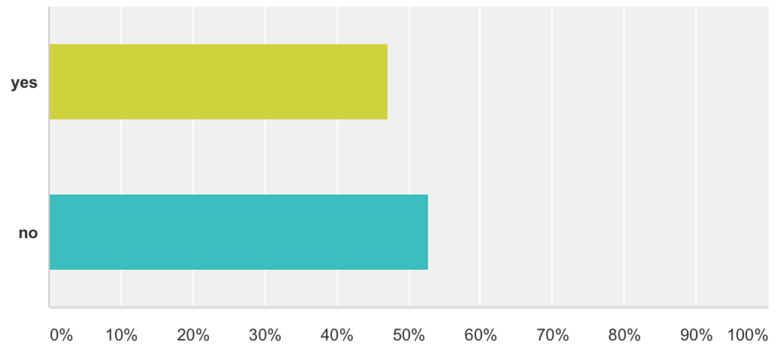
real	36
complex	16

Figure 2.5: What arithmetic are you using? Mark both if applicable.



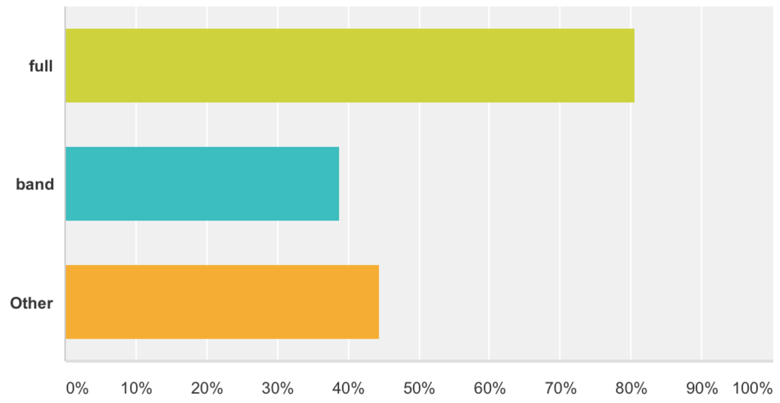
yes	13
no	23

Figure 2.6: Are you interested in extended precision, e.g., double-double or triple-float?



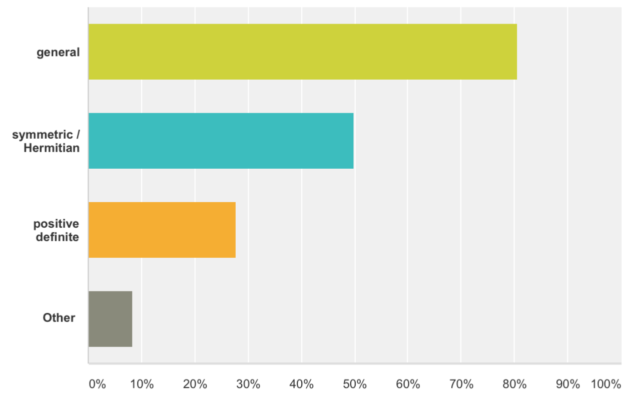
yes	15
no	17

Figure 2.7: Are you interested in lower precision, e.g., half precision (16-bit floating point).



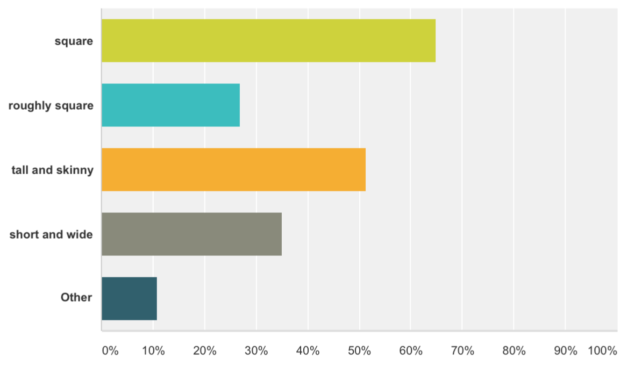
full	25
band	13
Other	14

Figure 2.8: What is the structure of your matrices? Mark all that apply.



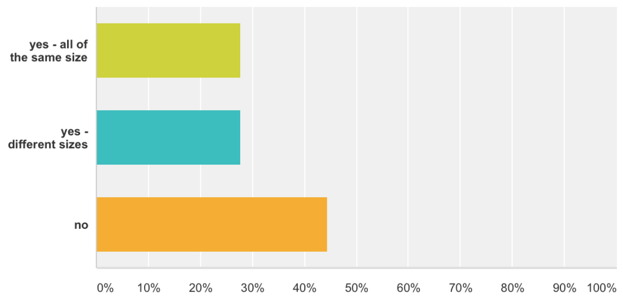
general	26
symmetric / Hermitian	16
positive definite	8
Other	3

Figure 2.9: What are the properties of your matrices? Mark all that apply.



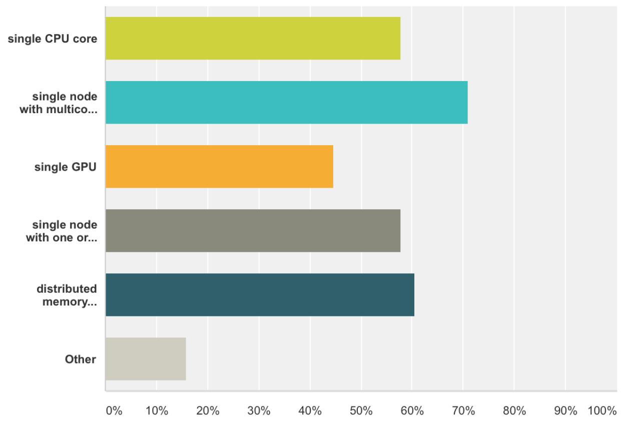
square	21
roughly square	10
tall and skinny	16
short and wide	13
Other	4

Figure 2.10: What is the typical shape of your matrices? Mark all that apply.



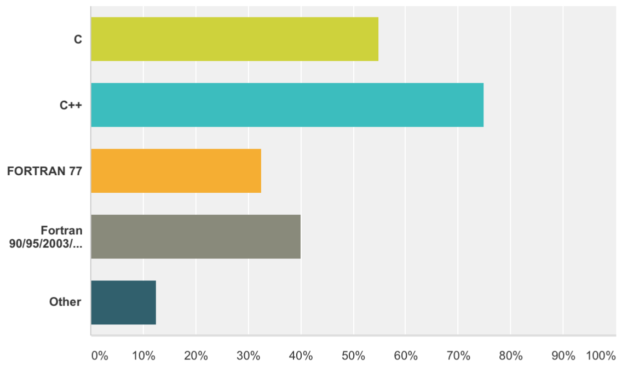
yes - all of the same size	10
yes - different sizes	7
no	14

Figure 2.11: Are you solving a large number of small independent problems? (matrices smaller than 500x500)?



single core	19
single node multicore	23
single GPU	15
single node +GPUs	18
dist. mem. [+GPUs]	22
Other	6

Figure 2.12: You need dense linear algebra routines for what target? Mark all that apply.



C	19
C++	26
FORTRAN 77	11
Fortran 90/95/2003/...	14
Other	4

Figure 2.13: Which API do you need? Mark all that apply.

2.2 Consequences for SLATE

This summary is based on the results of the survey, as well as follow up interaction with the applications teams. Here we summarize the main observations.

Impact: Dense linear algebra is ubiquitous in DOE ECP applications. 80% of respondents indicated reliance on LAPACK and BLAS, 35% reliance on ScaLAPACK, 20% reliance on PBLAS. The newer libraries, PLASMA and MAGMA have much smaller traction. While MAGMA has some adoption, the adoption of PLASMA is minimal. Half of respondents indicated that their applications rely completely or heavily on dense or band linear algebra operations. Only 20% of respondents indicated no need for such operations at all.

While traditional DOE applications, relying heavily on PDE solvers (fusion, combustion, wind turbines, stellar explosions), use little dense linear algebra, and mostly indirectly, there is a number of applications in quantum mechanics, computational chemistry, particle physics, material science, which rely heavily on dense linear algebra software (QMCPACK, NWChemEx, GAMESS, EXAALT). Naturally, dense linear algebra routines are used heavily by sparse direct solvers, e.g., FBSS (STRUMPACK).

Types of Problems: The needs of the ECP applications basically spans the coverage of BLAS and LAPACK. 75% of respondents indicated the need for BLAS, 50% indicated the need for linear solvers and eigenvalue solvers, 30% indicated the need for least squares solvers, singular value solvers, low rank approximations, and constructing an inverse of a matrix. There is also some need for updating and downdating capabilities.

80% of respondents deal with full matrices, 40% of respondents deal with band matrices. At the same time, 80% of respondents deal with general (non-symmetric) matrices, 40% deal with symmetric matrices, and 30% deal with symmetric positive definite matrices. Also, while in most cases the matrices are square (60%), there is also a strong need for operations on matrices that are tall and skinny, and on matrices that are short and wide (50% and 35% respectively). This, again, confirms the need for supporting the fairly wide range of matrix shapes and properties, close to the coverage of BLAS and LAPACK.

Target Hardware: There seems to be a universal agreement that all types of architectures should be supported, including homogeneous multicore systems, as well as heterogeneous, accelerated, systems, and support for multiple accelerators per node is desired. There also seems to be a universal agreement that distributed memory capabilities are a must, although we only heard a single request for actual exascale capabilities. The NWChemEx team expressed interest in a dense solvers capable of dealing with $O(1M)$ matrices.

Arithmetic and Precision: The majority of applications require double precision, while about half also use single precision. All need real arithmetic, while about half needs complex arithmetic. About a third indicated interest in extended precision (double-double or triple-float), while almost half indicated interest in lower precision, e.g., 16-bit half precision. This is a clear indicator that there is a strong need for providing flexibility going beyond the four basic precision supported by LAPACK, ScaLAPACK, PLASMA, and MAGMA.

Desired APIs: Basically, APIs for all common HPC programming languages are needed: C, C++, legacy Fortran, modern flavors of Fortran. At the same time, there is a strong interest in C++ interfaces. 75% of respondents indicated the need for a C++ API.

Batched Operations: There is a significant demand for batched dense linear algebra operations, i.e., operations on large numbers of small matrices. The two cases seem to be equally important: the case when all the matrices in the batch are of the same size, and the case when the batch contains matrices of different sizes. Obviously, this is a node-level capability.

In summary, the SLATE software needs to:

- serve as a replacement for BLAS/PBLAS and LAPACK/ScaLAPACK,
- support distributed memory systems with accelerators,
- provide a C++ API in addition to traditional APIs,
- facilitate the use of other precisions then the traditional set of four.

In the course of the project, it is also desired to:

- standardize the C++ APIs of BLAS and LAPACK,
- standardize the API for batched BLAS and LAPACK operations.

CHAPTER 3

Hardware Technology Trends

3.1 Upcoming Machines

The Collaboration of Oak Ridge, Argonne, and Livermore (CORAL) is a joint procurement activity among three of the Department of Energy's National Laboratories launched in 2014 to build state-of-the-art high-performance computing technologies that are essential for supporting U.S. national nuclear security and are key tools used for technology advancement and scientific discovery [61]. The Argonne Leadership Computing Facility (ALCF) will contract with Intel and Cray to build "pre-exascale" systems, while Lawrence Livermore National Laboratory (LLNL) and Oak Ridge Leadership Computing Facility (OLCF) will contract with IBM. This configuration of systems will enable explorations of architecture diversity along the path to exascale computing [63, 69].

The ALCF CORAL system, named Aurora, is planned on the foundation of Intel's HPC scalable system framework. Aurora is designed to provide peak performance of 180 PetaFLOP/s from >50,000 nodes while consuming 13 MW of power [69]. Argonne and Intel have also provide an interim system, called Theta, which is enabling ALCF users to transition their applications to the new technology [63, 68].

The LLNL CORAL system, named Sierra, is planned to provide 120-150 petaflop/s peak. The Sierra system will include compute nodes (POWER Architecture Processor, NVIDIA Volta, NVMe-comaptible PCIe 800 GB SSD, greater than 512 GB DDR4 + HBM "high bandwidth memory", and coherent shared memory), compute racks (standard 19-inch with warm-water cooling), and the compute system with be 2.1-2.7 PB memory, 120-150 petaflop/s, and 10 MW). The Global Parallel File System will have 120 PB usable storage and 1.0 TB/s bandwidth [73].

The OLCF CORAL system, named Summit, is planned to deliver more than five times the computational performance of OLCF's Titan's 18,688 nodes, using only approximately 3,400 nodes. Each Summit node will contain multiple IBM POWER9 CPUs and NVIDIA Volta GPUs all connected together with NVIDIA's high-speed NVLink and a huge amount of memory. Each node will have over half a terabyte of coherent memory (HBM + DDR4) addressable by all CPUs and GPUs, plus an additional 800 gigabytes of NVRAM [39].

3.2 Processing

3.2.1 GPUs

Graphical Processing Units adopt a many-core architecture. GPUs pack typically thousands of very lightweight processing cores, rather than tens of large powerful cores (as in modern multicore CPUs and the Xeon Phi architecture). Generally, a GPU thread executes slower than a CPU thread. Such slow single-thread execution is compensated by the ability to execute orders of magnitude more concurrent threads than a modern CPU. Therefore, GPUs usually outperform CPUs in executing kernels that include large amounts of parallelism, and where throughput matters more than latency.

Since the Fermi product line, GPU compute cards have always consisted of tens of Streaming Multiprocessors (SMs). Each SM consists of single precision CUDA cores, double precision units, texture units, warp schedulers, special function units, and fast memory levels (register file, shared memory, and sometimes a constant read-only memory). Through the Fermi, Kepler, and Maxwell architectures, the numer of SMs was relatively low (typically 8-15), and the main memory was an off-chip GDDR memory. The Pascal architecture [1] brought some drastic changes to these two aspect. First, a Pascal GPU packs 56-60 SMs, but the number of cores per SM is relatively low (e.g. 192 cores in Kepler vs. 64 cores in Pascal). In addition, the main memory is now a stacked memory architecture (HBM2) that brings substantial improvement in memory bandwidth (e.g. $\sim$250 GB/s for Kepler vs. 720 GB/s for Pascal). Throughout the generations, several other features have been added, such as atomic operations, dynamic parallelism, Hyper-Q, warp shuffle instructions, and improved power efficiency.

The latest commercially available GPU adopts a Pascal architecture [1]. A Pascal GPU (P100) has brought several architectural enhancements that are summarized:

1. Improved performance: the P100 GPU is capable of delivering up to 5.3/10.6 Tflop/s of performance. For double precision, this is more than $4\times$ improvement against the last FP64 capable GPU (Kepler).
2. Half precision arithmetic: The Pascal P100 GPU is the first NVIDIA GPU to support half precision arithmetic, with up to 21.2 Tflop/s. This new floating point standard is of particular importance for deep learning and AI applications.
3. High Bandwidth Memory (HBM2): the P100 chip incorporates a stacked memory architecture, with up to 720 GB/s bandwidth. Unlike the previous generations, this is the

first NVIDIA GPU to support ECC on the hardware level. Previous generations enable a software ECC, which consumes about 12-15% of the available memory bandwidth on average.

4. NVLink: a new high speed interconnect that can deliver at least $5\times$ the speed of the PCIe interconnect.
5. The P100 GPU is the first to support FP64 atomic operations on the hardware level.
6. Compute preemption at an instruction level granularity, rather than a thread block granularity in the previous generations.
7. A $2\times$ improvement in the RDMA bandwidth, which is fundamentally important for multi-GPU configurations.

The anticipated Volta GPU brings even more performance and power efficiency than the P100 GPU. While a detailed description of the V100 architecture is yet to come, these are the features announced for the upcoming GPU [2]:

1. Improved performance: the Tesla V100 GPU is at least $1.4\times$ faster than the P100 GPU in many computational workloads and benchmarks. This includes matrix multiplication (FP32 and FP64), Fast Fourier Transform (FFT), and the STREAM benchmark.
2. Faster interconnect: the Volta NVLink delivers up to 300 GB/s, which is almost twice as fast as the bandwidth of the Pascal NVLink.
3. A new ISA with double the number of warp schedulers (vs. Pascal). The volta GPU incorporates a larger and faster unified L1 cache/shared memory. Unlike Pascal, which has a separate 64KB shared memory and a slower 24KB L1 cache, the V100 GPU has a unified 128KB L1 cache/shared memory. The L1 cache is, therefore, as fast as shared memory. The latter is also configurable up to 96KB. The V100 GPU also has 6MB of L2 cache (against 4MB on the P100 GPU).
4. Independent thread scheduling: All previous GPUs had a single program counter (PC) and stack for a warp of 32 threads. As per the Volta GPU, each thread has its own PC and stack. A new synchronization mechanism among divergent threads in the same warp is also supported.
5. As a result of the independent thread scheduling, the volta GPU, along with CUDA 9.0, deprecates the previous shuffle and compare intrinsics, and replaces them with other intrinsics that synchronizes across a warp.
6. Tensor acceleration: the V100 GPU brings 8 tensor cores per multiprocessor. These cores perform mixed precision matrix math (FP16/FP32) with significant speedups against the P100 GPU. With up to 120 Tflop/s, the tensor cores are of particular importance for deep learning applications.
7. The Volta GPU comes with CUDA 9.0, which brings many new functionalities, such as cooperative thread groups, synchronization across thread blocks, and the elimination of implicit warp synchronization.

3.2.2 Xeon Phi

Knights Landing (KNL) is the codename for the second-generation of Intel's Xeon Phi many integrated core (MIC) architecture. The Theta system is a pre-exascale machine being installed and put into early production at Argonne National Lab by the Argonne Leadership Computing Facility (ALCF). Theta utilizes Intel Knights Landing along with the Cray Aries interconnect via the XC40 supercomputer architecture. The ALCF Theta XC40 system achieves a nearly 10 PF peak with 2,624 Nodes. Theta contains the 64 core 7230 KNL variant. The 7230 KNL chip has 64 cores that are organized into 32 tiles, with 2 cores per tile, connected by a mesh network and with 16 GB of in-package multichannel DRAM (MCDRAM) memory. The core is based on the 64-bit "Silvermont" Atom Core (1.40 GHz) which has 6 independent out-of-order pipelines, two of which perform floating point operations [68].

The 7230 and 7250 KNL variants utilizes 14 nm lithography similar to the "Broadwell" Xeon E5 and E7 server processors to achieve 3.05 TFLOPS Peak Double Precision. The 68 cores on the 7250 KNL chip are also based upon the "Silvermont" microarchitecture and support four execution threads. Both the 7230 and 7250 KNL variants have two AVX512 vector processing units per core. The cores are tiled in pairs that share 1MB of L2 memory. The tiles of both variants are linked to each other using 2D mesh interconnect, that also connects to the 384 GB DDR4-2400 memory (115.2 GB/s) through two controllers. The cores are also connected to 16 GB MCDRAM providing up to 490 GB/s of sustained bandwidth through the 2D mesh interconnect. The memory bandwidth per KNL core is approximately 11 GB/sec for small thread counts, while its predecessor the Knights Corner only provided 3.6 GB/sec [54].

Knights Hill (KNH) is the codename for the third-generation of Intel's Xeon Phi MIC architecture. Little has been publicly announced about the Knight's Hill Processor beyond the plan to manufacture it with a 10 nm lithography process (SC 14) [69]. ALCF's Aurora, a 180-petaflop supercomputer that is planned to be built for Argonne National Lab in 2018 was originally announced as utilizing KNH processors as part of the Collaboration of Oak Ridge, Argonne, and Livermore (CORAL) joint procurement activity [63]. The US Department of Energy fiscal year 2018 budget request is not specific regarding the Aurora supercomputer [62].

3.2.3 POWER

The Summit is Oak Ridge Leadership Computing Facility's (OLCF's) next flagship supercomputer. It will consists of 3,400 nodes, each featuring 2 IBM POWER9 CPUs and 6 NVIDIA Volta GPUs connected by NVLink and 512GB high bandwidth memory addressable by both CPUs and GPUs and 800GB of NVRAM.

For developers transitioning to use Summit, OLCF provides Summitdev with node architectures one generation earlier than Summit, the 2 IBM POWER8 CPUs with NVLink and 4 NVIDIA Tesla P100 GPUs. The IBM POWER8 [74] is a RISC microprocessor from IBM fabricated using IBM's 22-nm technology. Compared to previous generation IBM POWER7 processors the POWER8 processor improved both in single thread performance and in the number of cores. The IBM POWER8 processor feautures up to 12 cores per socket, 8 threads per core, 32 KB instruction cache, 64 KB L1 data cache, 512 KB L2 cache, and 8 MB L3 cache, per core. Each core can issue up to 10 instructions per cycle, and complete 8 instructions per cycle. Functional unit wise, the POWER8 has two independent fixed-point units (FXU), two independent load-store units (LSU) plus two more load units, and two independent floating-point vector/scalar units (VSU). The maximum double-precision floating-point issue rate is 4 fmadds per cycle, single-precision 8 fmadds per cycle. The SIMD width is 2 for double precision and 4 for single precision. Thus the peak double precision performance is Freq $\times$ 8 $\times$ #cores. For a 3.5 GHz frequency 20-cores node the peak DP performance is 560 GFLOPS.

The upcoming IBM POWER9 CPUs [72] will be fabricated using 14nm FinFET technology. It will come in 4 variants with either 1) 2 sockets optimized (Scale-Out, SO) or multiple sockets optimized (Scale-Up, SU) configurations, 2) 24 SMT4 cores, or 12 SMT8 cores. Compared to a POWER8 chip with the same SMT, frequency, and core count configuration, the POWER9 socket achieves 1.5x floating point performance, 2.25x graph analytics performance, and something between 1.5x and 2.25x for commercial, integer, scripting, and business intelligence workloads [79]. For HPC workloads, POWER9 is also a powerful acceleration platform equipped with NVLink 2.0 and various high bandwidth low latency interconnects (PCIe G4, CAPI 2.0, and New CAPI) to connect to accelerators such as ASIC/FPGA.

3.2.4 ARM

Recently many ARM vendors are trying to bring server class ARM chips to challenge Intel's Xeon in datacenter servers, and Japan, China, and Europe seem to be interested in a ARM based exascale supercomputer. Products in the server class ARM processors include Cavium with its high core count ThunderX and ThunderX2 lines. The current production chips, ThunderX, features up to 48 cores per socket in a 2 socket configuration, is fabricated using 28nm process technology, and clocked at up to 2.5 GHz (though 2.0 GHz is mostly found). The next generation ThunderX2 promises to have 2-3X improvement in performance and will be etched using 14nm FinFET process. Architectural change from ThunderX to ThunderX2 probably includes bigger caches, out-of-order pipelines, and more cores (56) per socket, clocked at 3.0 GHz. ThunderX2 is also supposed to be much more power efficient than ThunderX, and twice as high memory bandwidth.

To cater for the HPC market, the ARMv8-A architecture was exteneded with vector instructions. This was announced by ARM at the HotChips'16 conference [76]. The specifics of the Scalable Vector Extension (SVE) were later specified in more detail [77]. The SVE significantly enchances the vector processing capabilities of AArch64 execution in ARM architecture in a flexible way such that unmodified binary code can efficiently run on future CPUs with longer vector lengths (128-2048 bits in 128 bits increments). This is called Vector Length Agnostic programming model. Compared with previous media-processing focused SIMD instructions (aka the ARM NEON, or ARMv7 Advanced SIMD), the SVE introduces scalable vector lengths, gather load and scatter store, per-lane predication, and some other features to make it a better compiler target and allow increased parallelism extraction for HPC workloads. One particular interesting property in SVE is its vector length agnostic which increases its future proofness. Traditionally SIMD is incorporated into an ISA whenever the vector length is increased. We have seen this in x86 ISA with a handful of extensions: MMX, 3DNow!, SSE, SSE2, SSE3, SSSE3, SSE4, AVX, AVx2, AVX512). The SVE does not have to be revised every time the vector length is increased. In addition, the portability across different microarchitecture is improved with CPUs featuring different vector lengths with different tradeoffs between cost, power, area, and performance. Besides this, SVE has many other advanced vector processing functions that makes vectorization more applicable and efficient. As a result, SVE can achieve better speedups than NEON even with the same vector lengths for many applications, notably dense linear algebra, and allows CPU designers to find their optimum vector lengths depending on individual objectives and constraints. The SVE will be supported by lead partners of ARM such as Fujistu for its Post-K supercomputer. It is unclear whether Cavium's ThunderX2 will support SVE.

3.3 Communication

3.3.1 NVLINK

NVLink is a high memory bandwidth technology developed by NVIDIA in response to the slow communication via the PCIe interconnect. Current PCIe technology (Gen. 3.0, 16 lanes) has a theoretical peak bandwidth of 16 GB/s. With a significant portion of the computing power being delivered by the coprocessor, communication via this port quickly becomes the bottleneck when running distributed codes on a GPU-accelerated cluster. NVIDIA's NVLink-enabled P100 GPUs feature 4 NVLink connections, each providing a peak bandwidth of 20 GB/s per direction. The NVLink technology can be used in different configurations. Connecting all NVLink lanes to the CPU augments the host-accelerator bandwidth by another 80 GB/s per direction, which makes GP100 in total 160 GB/s of bidirectional bandwidth. Alternatively, a Peer-to-Peer configuration uses some of the lanes to improve the GPU-to-GPU communication. NVLink also boasts up to 94% bandwidth efficiency.

Aside from the massive bandwidth of NVLink, it is also designed to enable the clustering of GPUs and CPUs for them to appear as a single computing unit. NVLink enables this abstraction by supporting load/store semantics [40] that allows programmers to directly read/write peer GPU's local memory and the CPU's host memory all in a common shared memory address space. Furthermore, remote atomic memory operations are supported on peer GPUs for fast synchronizations. Together with the unified memory space in CUDA 8 and page faulting hardware in GP100 GPU, NVLink thus pushes forward something like Symmetric Multiprocessing (SMP) capability for CPUs to GPU accelerators (also to include CPUs that support NVLink). In fact, the NVIDIA's DGX-1 server clusters two CPUs with eight P100 GPUs, with GPU$\leftrightarrow$CPU links being PCIe and GPU$\leftrightarrow$GPU link being NVLink.

3.3.2 InfiniBand

InfiniBand originated in 1999 from the merger of two competing technologies: Future I/O and Next Generation I/O. Companies participating in the initial development included: Compaq, Dell, HP, IBM, Intel, Microsoft, and Sun. The first specification of the architecture was released in 2000. The vision was that InfiniBand would replace PIC for the I/O and

Ethernet for the interconnect. This plan was disrupted by the burst of the doc-com bubble. Mellanox shipped the first 10 Gbit/s devices in 2001.

Currently, InfiniBand is the most commonly used interconnect for supercomputers. InfiniBand adapters and switches are made by Mellanox and Intel, and Oracle is also entering the market with its own hardware. The important feature, from the standpoint of HPC, is that InfiniBand provides RDMA capabilities for low CPU overhead. The current technology, InfiniBand EDR, provides ca. 24 Gbit/s of theoretical effective throughput per link ($1\times$). Links can be aggregated, and most systems use a $4\times$ aggregate, for the total theoretical effective throughput of close to 100 Gbit/s.

3.3.3 OmniPath

Omni-Path is an interconnection technology from Intel developed to address scaling weaknesses that are currently impeding the High Performance Computing. From the commercial standpoint, its main competition is InfiniBand. OmniPath has its roots in two acquisitions - acquisition of the InfiniBand technology from QLogic and acquisition of the Aries interconnect technology from Cray. As a result, Omni-Path combines the QLogic True Scale architecture, and its associated software stack, with high performance features of Cray's Aires. Notably, it is compliant with the Open Fabrics Alliance (OFA) stack for RDMA fabrics.

The current generation of Omni-Path products delivers 100 Gbit/s of bandwidth per port and port-to-port latencies comparable to EDR InfiniBand. Notably, Intel already ships Xeon Phi processors with integrated Omni-Path fabric. Intel also differentiates the technology from other RDMA fabrics by pointing to a set of unique features, including traffic management and robust error detection, e.g.:

Adaptive Routing monitors the network and selects the least congested path to rebalance the load. While this is not a unique solution, Intel points out advantages of its implementation, which is based on cooperation between the fabric manager and the switch ASICs.

Dispersive Routing distributes traffic across multiple paths, instead of sending all packets from source to destination via a single path, which promotes efficiency through redundancy, and better load balancing.

Traffic Flow Optimization breaks up variable length packets into fixed size containers for transmitting over the link. At the same time, higher priority containers can be inserted into the stream of lower priority containers, which allows to reduce the variation of latency experienced by high priority traffic in the presence of low priority traffic.

3.4 Memory

3.4.1 High Bandwidth Memory

Memory interfaces have been undergoing major changes during the last few years and a number of new technologies are becoming available in high performance computing architectures. The basic technologies in previous generation memory interface (DDR4) were defined in 2008, and even though there have been updates for power and performance, these changes have been incremental. The maximum bandwidth for 64-bit DDR4 is approximately 26 GB/s, which is be insufficient for many application on highly multi-core architectures.

High Bandwidth Memory (HBM) is a memory interface promoted by AMD, NVIDIA, and Hynix. HBM is a new type of memory architecture with low power consumption and ultra-wide communication lanes. It uses vertically stacked memory chips interconnected by microscopic wires called *through-silicon vias* or TSVs (Figure 3.1).

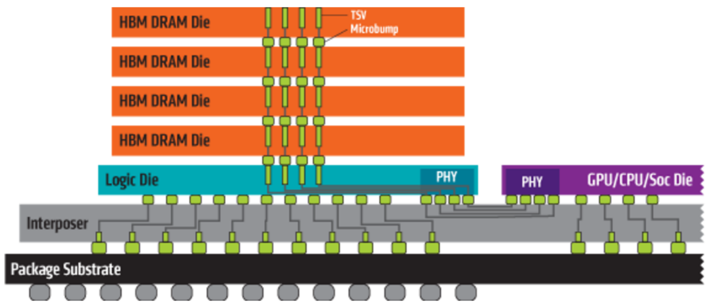


Figure 3.1: High Bandwidth Memory architecture (source: http://www.amd.com/en-us/innovations/software-technologies/hbm).

The primary purpose for HBM is for use in graphics cards and it is designed to serve as a replacement for GDDR5, which is the current graphics memory interface standard. The HBM interface is intended to decrease power consumption, to enable more dense circuitry for the higher bandwidth required today, and to allow the memory interface to be attached off-chip rather than requiring on-chip integration.

High Bandwidth Memory can achieve a bandwidth of 100+ GB/s per memory stack, whereas GDDR5 got up to 28 GB/s per chip. In terms of energy consumption, HBM has a substantially improved energy usage when measured in bandwidth/watt as compared to GDDR5. HBM achieves 35+ GB/s of bandwidth per watt, whereas GDDR5 achieves 10.66 GB/s of bandwidth per watt. The newer HBM-2 standard can achieve even better bandwidth and power efficiency.

3.4.2 Hybrid Memory Cube

The Hybrid Memory Cube (HMC) interface is managed by a consortium whose participants include Altera (Intel), ARM, IBM, Micron Technology, Open-Silicon, Samsung, SK Hynix, and Xilinx, along with a large number of adopter members.

Hybrid Memory Cube architecture consists of stacked memory chips that are bonded together using TSVs (through-silicon-vias) in conjunction with a high-speed logic layer (Figure 3.2).

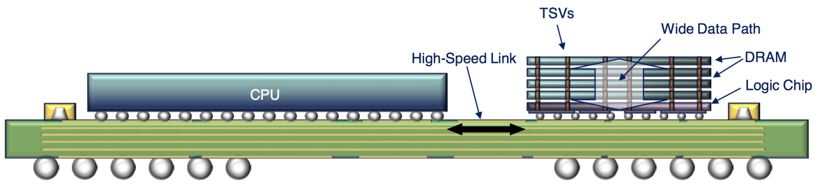


Figure 3.2: Hybrid Memory Cube architecture (source: http://wccftech.com/micron-hybrid-memory-cube-3-0-specification/).

One of the goals of HMC is to remove the duplicated control logic of modern memory systems, simplify the design, connect the entire stack in a 3D configuration, using a single control logic layer to handle the memory management. The logic layer in HMC controls all aspects of the memory, and the host memory control is simplified to handling requests and responses. The HMC logic layer provides error detection and management capabilities, atomic operations, reliability and availability features and scale-out device-chaining capabilities.

Table 3.1: Comparison of Memory Technologies.

Memory	HMC Gen3	HBM-2
Size	8 GB	8 GB
Max Bandwidth	480 GB/s	256 GB/s
Expandable	Yes, chain modules	No
Power	Higher	Lower
Target	HPC, networking	Graphics, networking, small form-factors
Benefits	High bandwidth; scalability; power efficiency	High bandwidth; scalability; power efficiency
Cost	High	Medium

A single HMC module can provide more than 15x the performance of a DDR3 module, utilizing 70 percent less energy per bit than DDR3 DRAM technologies, and is contained in 90 percent less space.

The Hybrid Memory Cube design has a higher power consumption than the High Bandwidth Memory design, but achieves a higher bandwidth. Putting the entire control logic into the HMC allows modules to be chained together to increase capacity.

3.5 Consequences for SLATE

The hardware technology trends, described in this chapter, have some dire consequences for dense linear algebra in general, and the SLATE project in particular. Here we summarize the most impactful developments:

Large Numbers of Cores: The number of CPU cores per node is going to be large. Right now, a single Xeon Phi is already at the level of 72 cores and one ThunderX ARMv8 system is at the level of 96 cores. As this trend continues, we will likely have hundreds of cores per node by the time we reach exascale. At the same time, it is still the rule of thumb that the best performance can be extracted from ScaLAPACK by running one process per core. It should be clear now that this is a completely unsustainable direction for any numerical software. While message passing is the paradigm of choice for the foreseeable future, node-level parallelism has to be addressed with some form of multithreading.

Omnipotent GPUs: GPUs are here to stay, and the GPU-accelerated machines are going to have virtually all of its performance on the GPU side. A single node of the Summit supercomputer is expected to have 40 TFLOPS of GPU performance. At the same time it is likely to have about 1 TFLOPS of CPU performance. This means that only 2.5% or raw performance comes from CPUs. A gap of such magnitude requires a fundamentally different approach to designing numerical software, as the process can no longer be framed in terms of "offloading" work to GPU. Instead, new packages have to be built from the ground up with a GPU-centric mindset.

Starved Communication: Communication is getting worse. Consider the following: The Titan supercomputer has node bandwidth of 6.4 GB/s and node peak performance of ca. 1.4 GFLOPS, while the Summit supercomputer is advertised at 23 GB/s of node bandwidth and 40 TFLOPS of node peak performance [5]. So, while the bandwidth increases about 3.6 times, the node peak performance increases more than 28 times, which means that the communication to computation ratio is about 8 times worse (basically an order of magnitude). This makes the bandwidth a very scarce resource. While various techniques can be used to deal with the latency, little can be done about the lack of bandwidth. This means that SLATE will have to be very conservative in its use of bandwidth, stay away from any techniques of dynamically rebalancing work, and rely on statically partitioning matrices among distributed memory nodes. Dynamic scheduling is only applicable at the node level.

Complex Node Memories: SLATE will have to deal with twofold memory complexity. First, a GPU-accelerated node is basically a globally-addressable distributed memory system. From the standpoint of programmability, memory traffic can be handled by a software coherency protocol. This does not change the fact that careful orchestration of data transfers will be required for good performance. Second, the introduction of 3D stacked memories creates an extra level of memory on the CPU side, with cache-like appearances, but no hardware cache coherency protocol. While low-level memory management may not be within the scope of SLATE, SLATE needs to utilize data layout that does not handicap memory motion to and from GPUs or between traditional RAM and 3D memory. Specifically, it is probably about time that the

ScaLAPACK matrix layout is retired, as it is not particularly friendly to transfers through memory hierarchies.

In summary, the SLATE software needs to:

- expose work in large chunks, to be able to saturate large numbers of CPU cores and/or multiple GPU devices per node,
- be extremely conservative it its use of network bandwidth, refrain from dynamic work migration between nodes,
- offer an alternative to the traditional matrix layout, which will streamline memory management and messaging.

CHAPTER 4

Software Technology Trends

The U.S. Department of Energy (DOE) has identified performance portability as a priority design constraint for pre-exascale as well as upcoming exascale systems [3, 4]. The development of portable across architectures parallel software can be provided through the use of standard APIs, e.g., OpenMP, OpenACC, and MPI, and standardized language features, like co-arrays and 'do concurrent' from the Fortran standard [70], or new parallelization features [52], proposed for inclusion in the C++17 standard, etc. However, as the standardizations and their efficient, high-performance implementations are a very slow process, many of these features remain inadequate for performance portability, especially as related to accelerator programming with heterogeneous compute capabilities and deep memory hierarchies. To address this, various programming environments and frameworks have been developed on top of standards as well, e.g., PaRSEC, Legion, DARMA, Kokkos, and RAJA, but still, the performance portability remains a major challenge. Therefore, there is need to understand, explore, and assess the current standards, as well as environments, in order to select the best programming model and practices, as related to performance portability, productivity, sustainability, and their trade-offs, for the development of the SLATE linear algebra library for exascale computing.

4.1 Standards

4.1.1 C++

Historically speaking, ScaLAPACK, PBLAS, and BLACS were written in a combination of FORTRAN 77 and C (ANSI and K&R versions). The complexities of modern hardware and software systems for HPC necessitated introduction of contemporary programming languages to ease the development process and shift the burden of common development tasks onto the software stack and the accompanying tool chain.

A quickly increasing set of scientific applications relies on a mix of programming languages and thus linking multiple language runtimes has become a common place. C++ has become a prevalent implementation language for large scientific software collections such as Trilinos and, as a consequence, became an indispensable part of large and scalable applications of great importance to the national defense and energy agenda.

As a somewhat arbitrary boundary, we choose to focus on the transition from C+03 to C++11. The former being mostly a bug-fix release of the C++98 standard that addressed a large number of defect reports. The latter, on the other hand, slipped a large aspirational deadline in 2010s and eventually landed almost a decade after the standard preceding it.

Even today, C++11 is not universally implemented by all the compilers. The prominent examples are vendor compilers that often substantially lag behind in features. More specifically, IBM's XLC tool chain is notoriously conservative and NVIDIA's compiler (from CUDA 8 as of this writing) still misses some features to make it C++11 complete. These are among the targeted compilers and therefore we have to take this situation into consideration.

A defensive strategy needs to be developed to pick and choose the C++ feature set that will be used throughout SLATE. As a mitigation strategy, we might consider a two-compiler strategy whereby both performance and programmer's productivity are addressed. As the project progresses, hopefully, support for modern features will improve, so workarounds can be replaced with more legitimate solutions.

In the following subsections we describe the modern C++ features most relevant to the development of SLATE.

Overloading

C++ allows multiple functions to have the same name, as long as they can be differentiated by argument types. For instance, a single `gemm` function, with versions for float, double, complex-float, complex-double, etc., instead of multiple type-specific variants: `sgemm, dgemm, cgemm, zgemm,` etc. This is crucial for templating, as all function calls must be generic.

Templates

C++ templates reduce the complexity of programming by implementing a routine once for a generic type, which can then be automatically instantiated for specific types such as single, double, half, or quad precision. Existing LAPACK, MAGMA, and PLASMA software involves either hand coding 4 versions (s, d, c, z) of each routine, or coding the double-complex version and using a search-and-replace script to crudely automate conversion to other precisions. Templates fully automate this process and ensure type safety.

When a template is instantiated for a specific type, all the operations and functions must apply to that type. In numerical software, this often involves adding no-op versions of functions. For instance, `conj(x)` when x is `double` simply returns x.

Traits

Traits is a technique to define types and parameters based on the type in templated code. For instance, the result of norm should be float for both `float` and `complex<float>`, as demonstrated in this example:

```
template< typename T >
class Traits
{
public:
    // by default, norm is same type: float => float, ...
    typedef T norm_type;
};

template< typename baseT >
class Traits< std::complex<baseT> >
{
public:
    // for complex, norm is base type: complex<float> => float, ...
    typedef baseT norm_type;
};

template< typename T >
typename Traits<T>::norm_type max_norm( int n, T* x )
{
    typename Traits<T>::norm_type norm = 0;
    for (int i = 0; i < n; ++i) {
        norm = std::max( norm, std::abs( x[i] ));
    }
    return norm;
}
```

Expression Templates

A simple C++ Vector class implementation would require intermediate temporary arrays to evaluate an expression such as:

```
Vector x(n), y(n);
x = 1.2*x + 2.0*y;
```

It would effectively become

```
tmp1 = 1.2*x;
tmp2 = 2.0*y;
```

```
tmp3 = tmp1 + tmp2;
x = tmp3;
```

Expression templates were developed as a techinque to evaluate these kinds of expressions efficiently, without any intermediate temporaries. It relies on lazy evaluation, that intermediate results are represented by meta-objects, and the actual operation is not performed until the assignment (=) operator. At the assignment, the compiler evaluates the meta-objects, usually inlining their code to effectively generate the "ideal" loop:

```
for (size_t i = 0; i < x.size(); ++i) {
    x[i] = 1.2*x[i] + 2.0*y[i];
}
```

with no intermediate temporary arrays.

Expression templates work well for Level 1 (vector) and Level 2 (matrix-vector) BLAS operations. Level 3 (matrix-matrix multiply) BLAS operations introduce more challenges. Aliasing becomes a major problem, for instance in the expression:

```
C = A*C;
```

updating C will produce erroneous results; a temporary is needed. It is difficult for the compiler to determine if aliasing will occur, and therefore whether a temporary is needed. It is also challenging to reach the peak performance for Level 3 BLAS with generic code. Hand-optimized code, possible in assembly or using hardware-specific intrinsics, generally has a large performance advantage over generic code. Multi-threaded expression templates is also challenging. Several BLAS libraries (uBLAS, MTL4, Eigen, etc.) have been developed around the idea of expression templates; a further evaluation of these is available in the accompanying *C++ Binding for BLAS and LAPACK* document.

Exceptions

Traditional linear algebra software such as LAPACK has relied on returning an info parameter with an error code. C++ allows throwing exceptions, which a parent context can catch. This can simplify error checking by grouping all the error checks together. Exceptions also prevent ignoring errors, as often happens with returned error codes—the exception must be caught somewhere, or it will propogate all the way up to main.

Existing C code often mises error handling after every function call. Especially mixed with allocation, this is error prone as all previous allocations must be freed:

```
double *A = NULL, *B = NULL;
A = malloc( lda*n*sizeof(double) );
if (A == NULL) {
    // handle error
}

B = malloc( ldb*n*sizeof(double) );
if (B == NULL) {
    free( A );  A = NULL;  // forgetting this causes a memory leak!
    // handle error
}

// computing using A and B

free( A );  A = NULL;
free( B );  B = NULL;
```

Using exceptions moves this error checking code to the end:

```
double *A = nullptr, *B = nullptr;
try {
    A = new double[ lda*n ];
    B = new double[ ldb*n ];
    // ... computing using A and B ...
}
catch( const std::bad_alloc& exception ) {
    // ... handle error
}
delete[] A;  A = nullptr;
delete[] B;  B = nullptr;
```

Even better in this example is to have a Matrix class that encapsulates the new/delete, using the common *Resource Acquisition Is Initialization (RAII)* paradigm, so that matrices are automatically deleted when exiting their context:

```
try {
    Matrix A( lda, n );
    Matrix B( ldb, n );
    // ... computing using A and B ...
}
catch( const std::bad_alloc& exception ) {
    // ... handle error
}
```

C++ exceptions can be implemented as "zero-cost exception", meaning no extra time cost when exceptions are *not* thrown, compared to code that simply aborts on error. There is added cost to the size of object code, which must encode how to unwind the stack when an exception occurs, and time to actually unwind the stack when an exception does occur. Hence, exceptions should be invoked rarely. For further details, see https://mortoray.com/2013/09/12/the-true-cost-of-zero-cost-exceptions/

Care must be taken with exceptions in multithreaded code and when interoperating between mutiple languages. C++11 introduced `std::exception_ptr` to facilitate such uses.

Value Initialization

As one of very few features added in C++03, it is a very commonly used feature that allows, among other things, providing a default constructor for user-defined objects. Trillinos packages, Epetra and Teuchos, use it for their BLAS and LAPACK wrappers.

Type Inference (C++11)

A significant contribution of C++11 is the decrease in verbosity by adding *type inference* functionality to the compiler by changing the semantics of the `auto` keyword and adding a new keyword: `decltype`. However, these are of limited use for functions that work with primitive data types and without templates. Another drawback is the compiler compliance, CUDA 8 with its `nvcc` compiler being a notable example.

```
std::vector< doube > x( 10 );

// verbose syntax
for (std::vector< double >::const_iterator iter = x.begin(); iter != x.end(); ++iter)
{ ... }
```

```
// simplified with auto
for (auto iter = x.begin(); iter != x.end(); ++iter)
{ ... }
```

Lambda Functions and Expressions (C++11)

Functional capabilities with generalized notion of a function and support for closures were a major addition of the C++11 standard. These facilities may be very useful for asynchronous operation and event-based processing that tend to fit well on a many-way parallel heterogeneous systems in a distributed processing context when network latency creates opportunities for out-of-order computing. This may be effectively combined with the parallel facilities of the C++ standard and allow for seamless passing of execution contexts.

NULL Pointer Constant (C++11)

Addition of an explicit and type-safe constant `nullptr` to use for pointers that are known to be uninitialized adds possibilities in the area of stricter typing and less error-prone code that relies on type-less values that can easily be confused with unsigned integer values. This resolves certain ambiguities with overloading, for instance:

```
void foo( int x ) {}
void foo( void* x ) {}

foo( 0 );       // calls foo( int )
foo( NULL );    // error: ambiguous
foo( nullptr ); // calls foo( void* )
```

Strongly Typed Enumerations (C++11)

Enumeration types are a useful feature that C++ inherited from its C pedigree. In C, enums are basically integers and can be exchanged freely with other integral types. C++98 added some type safety: you cannot implicitly convert from an int to an enum, or between different enums:

```
typedef enum { red, green } color_t;
typedef enum { left, right } side_t;
color_t a = red;    // ok
color_t b = 101;    // ok in C, error in C++
color_t c = left;   // ok in C, error in C++
```

However, you can implicitly convert from an enum to an int:

```
int d = red;        // ok in C and C++
```

Strongly typed enumerations in C++11 prevent this implicit conversion to int:

```
enum class Color { red, green };
Color e = Color::red;   // ok
int f = Color::red;           // error
int g = int( Color::red );    // explicit conversion ok
```

In both C and C++, an old-style enumeration may occupy an unknown size because the compiler is free to pick the implementation type based on the number of enumerated items. Strongly typed enumerations fix this problem by allowing the size to be specified; the default size is int:

```
enum class Color { red, green };                // sizeof(Color) == sizeof(int)
enum class ColorSmall : char { red, green };  // sizeof(ColorSmall) == sizeof(char)
```

Strongly typed enumerations also reduce name conflicts by scoping names. Here `red` in `color_t`, `Color::red`, and `ColorSmall::red` are all different enum values.

Memory Alignment Control

While it was possible to force alignment on pointers through non-portable means, the inclusion of alignment syntax is a welcome addition to the standard. As a result, it will be now possible to portably manipulate aligned memory and be able to trigger optimization levels that make it possible to use low-level features such as streaming loads and other vector instructions that only work with aligned addresses and often are not generated because the pointers are not guaranteed to be conforming and runtime checks are prohibitively expensive.

Implementation Specific Attributes

The C++11 ability to add attributes to various syntactic constructs gives a long-needed recognition of specifying features beyond the purview of the compiler and may reach the linking stages. This feature is often used in HPC codes, for example, in the form of weak symbols that allow the user to supply zero-cost tracing layer that is deactivated in production runs. It is to be determined if weak linking will be added to the standard.

Move Semantics (C++11)

The addition of rvalue references allows optimizing away a copy of temporary data into a simple move by swapping pointers. This greatly improves speed when, for instance, returned a large object from a function. However, traditionally BLAS and ScaLAPACK routines have taken all arrays as arguments, rather than returning arrays, so it is not clear that a benefit exists. The rvalue references can easily be conditionally compiled for older compilers, in which case code reverts to the old, slower behavior. In example below, class Foo has a move semantics constructor, `Foo( Foo&& tmp )`, and the copy-and-swap assignment operator inherits the move semantics from that constructor. For more details, see http://stackoverflow.com/questions/3106110/what-are-move-semantics and http://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom.

```
class Foo {
public:
    Foo( size_t size=0 ):
        m_size( size ),
        m_data( new double[size] )
    {}
    // expensive copy constructor: allocate and copy data from orig
    Foo( const Foo& orig ) {
        m_size = orig.m_size;
        m_data = new double[ m_size ];
        std::copy( orig.m_data, orig.m_data + m_size, m_data );
    }

#if (__cplusplus >= 201103)  // requires C++11
    // cheap "move semantics" constructor: move data from tmp
    Foo( Foo&& tmp ) {
        m_size = tmp.m_size;
        m_data = tmp.m_data;
        tmp.m_size = 0;
        tmp.m_data = nullptr;
    }
#endif

    // "copy and swap" idiom assignment operator
    Foo& operator = ( Foo copy ) {
        std::swap( a.m_size, b.m_size );
        std::swap( a.m_data, b.m_data );
        return *this;
    }

    ~Foo() {
        delete[] m_data;
    }

    size_t  m_size;
    double* m_data;
};

Foo factory();  // factory function defined elsewhere

void test() {
    // Without move semantics (C++98), this does expensive copy Foo( Foo& );
    // with move semantics (C++11), this does cheap move Foo( Foo&& )
    // (assuming Return Value Optimization (RVO) doesn't kick in).
    Foo C = factory();
}
```

Static Assertions (C++11)

`static_assert` enforces its condition at compile time, in contrast to `assert`, which enforces its condition at runtime. This helps to make more robust code, while not adding any overhead to the runtime cost. For instance:

```
static_assert( sizeof(int) == 4, "Requires␣4-byte␣int" );
```

Prior to C++11, static assertions can be hacked in various ways; Eigen has an example of this.

Smart Pointers (C++11)

In C++, smart pointers are abstract data types that simulate pointers while providing automatic memory management. They are intended to reduce bugs caused by the misuse of pointers, while retaining efficiency. Specifically, smart pointers prevent most memory leaks by making the deallocation automatic. I.e., an object controlled by a smart pointer is automatically destroyed when the last owner of an object is destroyed. Smart pointers also eliminate dangling pointers by postponing destruction until an object is no longer in use. In

C++, a smart pointer is implemented as a template class that mimics, by means of operator overloading, the behaviors of a traditional (raw) pointer, (e.g. dereferencing, assignment).

Consider the traditional declaration:

```
some_type* ambiguous_function();
```

There is no way to know whether the caller should delete the memory of the referent when the caller is finished with the information.

Alternatively, the following declaration:

```
unique_ptr<some_type> obvious_function1();
```

makes it clear that the caller takes ownership of the result, and the C++ runtime ensures that the memory for `*some_type` will be reclaimed automatically.

The following types of smart pointers are available:

unique_ptr explicitly prevents copying of its contained pointer, and provides the `std::move` function to transfer ownership to another `unique_ptr`

shared_ptr maintains reference counting ownership of its contained pointer. An object referenced by the contained raw pointer will be destroyed when and only when all copies of the pointer have been destroyed.

weak_ptr is a container for a raw pointer. It is created as a copy of a `shared_ptr`. The existence or destruction of `weak_ptr` copies of a `shared_ptr` have no effect on the `shared_ptr` or its other copies. After all copies of a `shared_ptr` have been destroyed, all `weak_ptr` copies become empty.

All smart pointers are defined in the `<memory>` header.

4.1.2 OpenMP

OpenMP – an abbreviation for Open Multi-Processing – is an application programming interface (API) based on compiler directives, some library routines, and environment variables for multiprocessing programming in C, C++, and Fortran. OpenMP is the de-facto standard API for shared memory parallel programming with widespread vendor support and a large user base. It implements multithreading, or the so-called "fork-join" parallel model of parallel execution. When any thread encounters a parallel construct, the thread creates a team of itself and zero or more additional threads and becomes the master of the new team. A set of implicit tasks, one per thread, is generated. The code for each task is defined by the code inside the parallel construct. A set of directives is provided to manage, synchronize, and assign work to threads that share data. Recently, with the adoption of OpenMP 4.0 and 4.5 (see below), the OpenMP shared memory programming model was extended to support task dependencies and accelerators, and this substantially changed the programming model from previous versions of the API.

Tasking Extensions

OpenMP 3.0 [11] introduced simple tasking that followed the Cilk model. OpenMP 4.0 [66] introduced data dependencies, allowing for proper expression of dataflow. OpenMP 4.5 further extended tasking capabilities, specifically added task priorities, which are critical from the performance standpoint. The basic concepts of OpenMP tasking include:

task A specific instance of executable code and its data environment, generated when a thread encounters a `task`, `taskloop`, `parallel`, `target`, or `teams` construct.

task region A region consisting of all code encountered during the execution of a task.

child task A task is a child task of its generating task region.

sibling task Tasks that are child tasks of the same task region.

task completion Task completion occurs when the end of the structured block associated with the construct that generated the task is reached.

task dependence An ordering relation between two sibling tasks: the dependent task and a previously generated predecessor task. The task dependence is fulfilled when the predecessor task has completed.

dependent task A task that because of a task dependence cannot be executed until its predecessor tasks have completed.

Tasks are generated when a thread comes across a task generating construct. Explicitly generated tasks are assigned to one of the available threads in the current team. Execution of a new task can be immediate or deferred until later, when threads are available and scheduling constraints are met. Threads are allowed to switch from one task to another at predefined *task scheduling points*. Tasks can be *tied tasks* or *untied tasks*. Suspended tied tasks must be resumed by the same thread. Suspended untied tasks can be resumed by a different thread. Task completion is guaranteed by the implicit barrier at the end of a parallel region. Task completion can be enforced in the middle of a parallel region by one of the task synchronization constructs: `taskwait`, `taskgroup`, or `barrier`.

The `depend` clause allows for expression of dataflow dependencies, i.e. scheduling constraints, between sibling tasks. Storage locations may be marked as `in`, `out`, or `inout`. If a storage location is marked as `out` or `inout` in one task, and marked as `in` in a subsequent task, then the latter task depends on the former task (cannot start execution before the former task completes). Also, if a storage location is marked as `in` or `out` in one task, and marked as `out` or `inout` in a subsequent task, then the latter task depends on the former task. All the different cases basically boil down to the three basic data hazards: Read After Write (RAW), *Write After Read* (WAR), and *Write After Write* (WAW).

Cancellation is a critical feature when dealing with the handling of exceptions. The `cancel` construct is a stand-alone directive that activates cancellation of the innermost enclosing region: `parallel`, `sections`, `for` or `taskgroup`. When a task encounters the `cancel` construct with the `taskgroup` clause, it cancels execution and skips over to the end of its task region, which implies completion of that task. Any other task, in the same group, that began execution, completes execution, unless it encounters the `cancellation point` construct. If it does, it also skips over to the end of the task region, which also implies completion. Any task that has not begun execution is aborted, again implying completion. The other cancellation clauses apply cancellation to the innermost enclosing region of the type specified. Execution continues at the end of the region. Threads check for cancellation at cancellation points. One important aspect of the `cancel` construct is that it cancels barriers, i.e., threads waiting in a barrier are released and skip over to the end of the canceled region. This can occur before all the expected threads reach the barrier.

Device Extensions

OpenMP 4.0 introduced new features that make it possible to run codes on both general-purpose multicore CPUs and accelerators, in a work-sharing fashion, under a single programming paradigm. The current standard, OpenMP 4.5, further improved acceleration features, and the proposed standard, OpenMP 5.0, extends them yet further. The basic concepts of the OpenMP acceleration model include:

host device The *device* on which the OpenMP program begins execution - basically the CPU.

target device A device onto which code and data may be offloaded from the host device - basically a GPU or a *leverage boot* Xeon Phi; generally referred to as an accelerator/coprocessor.

team A set of one or more threads participating in the execution of a parallel region. In the context of GPU execution, it is basically a *thread block*. It is not possible to synchronize across different teams over the lifetime of their existence.

league The set of thread teams created by a `teams` construct. In the context of GPU execution, it is basically a *block grid*.

The offload execution model of OpenMP is host-centric, meaning that the *host device* offloads `target` regions to *target devices*. The whole program is surrounded by an implicit parallel region executed on the host device. One or more devices may be available to the host device for offloading of code and data. Each device has its own distinct set of threads, which cannot migrate from one device to another.

The `target` construct is used to create a device data environment and to execute a code region on the device. Device id can be specified to select the device for execution if multiple devices are available. The runtime routine `omp_get_num_devices` can be used to determine the number of accelerators in the system. To handle the situation when there are no accelerators, the `target` construct can have an `if` clause. When an `if` clause is present and the expression evaluates to false, the device is the host. The `nowait` clause can be added to the `target` construct to allow for asynchronous execution. Also, a `target` region is implicitly enclosed in a target task region, and the `depend` clause can be added to specify the data flow dependencies of the implicit task generated for the target. The `declare target` construct can be used to declare a function as device function, basically flagging it as an accelerator/coprocessor kernel.

The `map` clause associates the current data environment on the host with the device data environment. Data creation and movement is controlled by the `to`, `from`, `tofrom`, and `alloc`, `release`, and `delete` attributes. The `target data` construct can be used to create a device data environment that is persistent across multiple `target` execution regions. The `is_device_ptr` clause is used to indicate that an item is a device pointer already and that it should be used directly. The `target update` clause can be used to update the host data with the corresponding device data, or vice versa, within one data region. The clause refreshes host data with the device data for the `from` clause, and device data with host data for the `to` clause. Two standalone directives, `target enter data` and `target exit data`, allow for mapping and unmapping data items to the device data environment.

Several runtime routines are also available to manage memory on target devices, including routines to allocate, free, and copy device memory, as well as routines to control mapping of device pointers to host pointers:

omp_target_alloc Allocates memory in a device data environment.

omp_target_free Frees the device memory allocated with `omp_target_alloc`.

omp_target_is_present Tests whether a host pointer has corresponding storage on a given device.

omp_target_memcpy Copies memory between any combination of host and device pointers.

omp_target_memcpy_rect Copies a rectangular subvolume from a multi-dimensional array to another multi-dimensional array.

omp_target_associate_ptr Maps a device pointer, which may be returned from `omp_target_alloc` or implementation-defined runtime routines, to a host pointer.

omp_target_disassociate_ptr Removes the associated pointer for a given device from a host pointer.

SIMD Extensions

The `simd\lstinline` construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions). The `simd` directive places restrictions on the structure of the associated *for-loops*. Specifically, all associated *for-loops* must have *canonical loop form*. The canonical form allows the iteration count of all associated loops to be computed before executing the outermost loop. The computation is performed for each loop in an integer type [66, 67]. The basic terminology of the OpenMP `simd` concept include :

SIMD instruction A single machine instruction that can operate on multiple data elements.

SIMD lane A software or hardware mechanism capable of processing one data element from a SIMD instruction.

SIMD chunk A set of iterations executed concurrently, each by a SIMD lane, by a single thread by means of SIMD instructions.

All loops associated with the construct must be perfectly nested; that is, there must be no intervening code nor any OpenMP directive between any two loops. The associated loops must be structured blocks.

A SIMD loop has logical iterations numbered 0,1,...,N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would be executed if the associated loop(s) were executed with no SIMD instructions. If the `safelen` clause is used then no two iterations executed concurrently with SIMD instructions can have a greater distance in the logical iteration space than its value. The parameter of the `safelen` clause must be a constant positive integer expression. If used, the `simdlen` clause specifies the preferred number of iterations to be executed concurrently. The parameter of the `simdlen` clause must be a constant positive integer. The number of iterations that are executed concurrently at any given time is implementation defined. Each concurrent iteration will be executed by a different SIMD lane. Each set of concurrent iterations is a SIMD chunk. Lexical forward dependencies in the iterations of the original loop must be preserved within each SIMD chunk [66, 67].

The declare `simd` construct can be applied to a function (C, C++ and Fortran) or a subroutine (Fortran) to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation in a SIMD loop. The declare `simd` directive is a declarative directive. There may be multiple declare `simd` directives for a function (C, C++, Fortran) or subroutine (Fortran) [66, 67].

Example Codes

This section presents example codes in C and C++ to highlight the use of OpenMP features discussed in the preceding sections.

To motivate an example of task-parallelism a serial form of the divide and conquer algorithmic technique for summation is shown in Figure 4.1. The code snippet in Figure 4.1 is based upon the Wikibooks example [82].

```c
float sum(const float *a, size_t n)
{
    // base cases
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return *a;
    }

    // recursive case
    size_t half = n / 2;
    return sum(a, half) + sum(a + half, n - half);
}
```

Figure 4.1: Basic divide and conquer summation algorithm.

Recognizing that the recursion in the divide and conquer summation consists of parallel tasks, rather than data parallelism, prompts the implementation of a task-recursive version, such as shown in Figure 4.2. The first pragma, `#pragma omp parallel`, prepares all the threads in the pool to execute the next code block and defines a `parallel` region. The second pragma, `#pragma omp single nowait`, utilizes the `single` directive to cause all threads but one to skip the next code block. The `nowait` clause disables the implicit barrier associated with the `single` directive and allows the threads that skip the next code block to move ahead to the barrier that concludes the `parallel` region.

Two tasks are used to acomplish the recursion, `#pragma omp task shared(x)`, for the first half of the float array, and `#pragma omp task shared(y)`, for the second half. The `task` construct defines an explicit task, which may be executed by the encountering thread, or deferred for execution by any other thread in the pool. Each task uses a `shared` clause to declare a variable shared with another task. Declaring these variables as shared ensures that values stored in them will persist after tasks complete. The last pragma, `#pragma omp taskwait`, causes execution to wait until all tasks have completed before combining the recursive results.

```
float sum(const float *a, size_t n)
{
    // base cases
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return 1;
    }

    // recursive case
    size_t half = n / 2;
    float x, y;

    #pragma omp parallel
    #pragma omp single nowait
    {
        #pragma omp task shared(x)
        x = sum(a, half);
        #pragma omp task shared(y)
        y = sum(a + half, n - half);
        #pragma omp taskwait
        x += y;
    }
    return x;
}
```

Figure 4.2: Task-recursive divide and conquer summation algorithm.

Next, the common example of adding two arrays, a and b, then storing the result in a third, c, is utilized to examine how code can be offloaded to a device. The `target` construct is used to specify the region of code that should be offloaded for execution onto the target device as show in Figure 4.3 [53]. The construct also creates a device data environment by mapping host buffers to the target for the extent of the associated region. The `map` clauses of the target construct specify data movement from host to device before execution of the offloaded region, and device to host after execution of the offloaded region.

In Figure 4.3, the `\#pragma omp target` construct initializes the the target region. A target region begins as a single thread of execution. When a target construct is encountered, the

```
float a[1024];
float b[1024];
float c[1024];
int size;

void vadd_openmp(float *a, float *b, float *c, int size)
{
    #pragma omp target map(to:a[0:size],b[0:size],size) map(from: c[0:size])
    {
        int i;
        #pragma omp parallel for
        for (i = 0; i < size; i++)
            c[i] = a[i] + b[i];
    }
}
```

Figure 4.3: Offloading the task-recursive divide and conquer algorithm.

target region is executed by the implicit device thread and the encountering thread on the host waits at the construct until the execution of the region completes. If a target device is not present, or not supported, or not available, the target region is executed by the host device. The variables a, b, c, and size initially reside in host memory. Upon encountering a target construct:

- Space is allocated in device memory for variables a[0:size], b[0:size], c[0:size], and size.
- Any variables annotated `to` within a `map` clause are copied from host memory to device memory.
- The target region is executed on the device. The `#pragma omp parallel for` is used to distribute iterations of the for loop across the device's thread pool.

When exiting a target construct: Any variables annotated `from` within a `map` clause are copied from device memory to host memory.

To further motivate an example use of device targets in the context of an application, a basic "escape time" algorithm for generating a Mandelbrot set is shown in Figure 4.4. The code snippet in Figure 4.4 is based upon the Wikipedia example [83]. For each x, y point in a rectangular plot area $(ImageWidth \times ImageHeight)$ an iterative calculation is performed. The x and y locations of each point are used as starting values of the iterating calculation. The real and imaginary values are checked during each iteration to see whether either have reached a critical "escape" condition. Because no complex number with a real or imaginary values greater than 2 can be part of the set, a simple escape condition is to stop iterations when either coefficient exceeds 2. Visual representations of the Mandelbrot set may be produced by translating the number of iterations required to escape from each pixel's x, y location to a color pallet.

One way of exploiting the data and task parallelism in the simple Mandelbrot set generation algorithm shown in Figure 4.4 is to offload the iterative calculation work load for each x, y point to a target device, as shown in shown in Figure 4.5. Several directives and clauses are utilized to accomplish the offload to a single `target device` enumerated as device 0.

```
int32_t ImageWidth = 1024;
int32_t ImageHeight = 1024;
uint32_t max_iter = 1000;
uint32_t in_vals[ImageHeight][ImageWidth];
uint32_t count[ImageHeight][ImageWidth];

for (int32_t y = 0; y < ImageHeight; ++y) {
  for(int32_t x = 0; x < ImageWidth; ++x) {
    uint32_t iteration = 1;
    fcomplex z = in_vals[y][x];
    for (int32_t i = 0; i < max_iter; i += 1) {
      z = z * z + c;
      int t = cabsf(z) < 2.0f;
      iteration += t;
      if (!t) { break;}
    }
    count[y][x] = iteration;
  }
}
```

Figure 4.4: Serial implementation of a Mandelbrot set generation algorithm.

Similarly to the previous example, the `#pragma omp target device(0)` initializes the target region, in this code snippet it is specifically for device 0. A target region also begins as a single thread of execution. When a target construct is encountered, the target region is executed by the implicit device thread and the encountering thread on the host waits at the construct until the execution of the region completes. If a target device is not present, or not supported, or not available, the target region is executed by the host device. The `mapto:in_vals)` clause designates that on entry to the target region `in_vals` will be copied from the host to device(0). The `map(from:count)` clause designates that on exit from the target region `count` will be copied from device(0) to the host.

`#pragma omp parallel` prepares all the threads in device(0)'s pool to execute the next code block and defines a `parallel` region. `#pragma omp for schedule(guided)` distribute iterations of the `for` loop across the the thread pool according to the `schedule(guided)` clause. Guided scheduling uses OpenMP's internal work queue to assign large chunks of loop iterations to threads at first and then decrease the chunk size to better handle load imbalances between iterations. The optional chunk parameter specifies the minimum size chunk to use. By default the chunk size is approximately loop_count/number_of_threads.

The declare target construct specifies that variables and functions are mapped to a device. Each function specified in a declare target region must have definitions for both the host and target device. In the code snippet, shown in Figure 4.5, the function mandel is called from within a target region. It's prototype must be placed in a declare target region.

The `#pragma omp simd` construct enables the execution of multiple iterations of the associated loops concurrently by means of SIMD instructions. The `safelen(16)` clause is used to specify that two iterations, executed concurrently with SIMD instructions, will not have a greater distance in the logical iteration space than the value (16). The parameter of the `safelen` clause must be a constant positive integer expression. The `simdlen(16)` clause specifies the preferred number of iterations to be executed concurrently. The parameter of the `simdlen` clause must be a constant positive integer. The number of iterations that are executed concurrently at any given time is implementation defined. Each concurrent iteration will be executed by a different SIMD lane. Each set of concurrent iterations is a

SIMD chunk. Lexical forward dependencies in the iterations of the original loop must be preserved within each SIMD chunk.

```
int32_t ImageWidth = 1024;
int32_t ImageHeight = 1024;
uint32_t max_iter = 1000;
uint32_t in_vals[ImageHeight][ImageWidth];
uint32_t count[ImageHeight][ImageWidth];

#pragma omp declare target
#pragma omp declare simd simdlen(16)
uint32_t mandel(fcomplex c)
{ // Computes number of iterations that it takes
  // for parameter c to escape the mandelbrot set
  uint32_t iteration = 1; fcomplex z = c;
  for (int32_t i = 0; i < max_iter; i += 1) {
    z = z * z + c;
    int t = cabsf(z) < 2.0f;
    iteration += t;
    if (!t) { break;}
  }
  return iteration;
}
#pragma omp end declare target

#pragma omp target device(0) map(to:in_vals) map(from:count)
#pragma omp parallel
{
  #pragma omp for schedule(guided)
  for (int32_t y = 0; y < ImageHeight; ++y) {
    #pragma omp simd safelen(16)
    for(int32_t x = 0; x < ImageWidth; ++x) {
      count[y][x] = mandel(in_vals[y][x]);
    }
  }
}
```

Figure 4.5: Parallel implementation of a Mandelbrot set generation algorithm highlighting exemplar use of the `target` construct with the `map` clause and the `simd` construct with the `safelen` clause.

4.1.3 OpenACC

OpenACC – an abbreviation for *Open Accelerators* – is a standard of compiler directives for parallel programming on heterogeneous CPU/GPU systems. Announced in 2011 with its first *OpenACC 1.0* release [64], OpenACC has continued growing in functionality as evident through its *OpenACC 2.0* (June 2013) and *OpenACC 2.5* (October 2015) releases [65]. The goal of these developments has been to provide the mechanisms for mapping parallelism - simply annotated by directives in a single source - to different hardware. Thus, to enable efficient mapping, powerful new features have been added to OpenACC, including nested parallelism, atomics, ways to target multiple devices, unstructured data regions (to retain data on accelerators between multiple OpenACC compute regions), asynchronous data moves, queue management, as well as profiling and tracing interfaces. Most of the communities working on these extensions also share ideas and collaborate with OpenMP, which also has been growing to currently support accelerators offload, as detailed in Section 4.1.2. This inevitably raises the question of which one to use, or both, or whether OpenACC and

Table 4.1: The most significant features introduced in the OpenACC standard since the standard's introduction.

OpenACC 1.0 (2011)	OpenACC 2.0 (2013)	OpenACC 2.5 (2015)
Wait directive	Routine directive	Asynchronous data transfer
Cache directive	Complex data lifetimes	Queue management routines
Host data construct	Nested paralellism	Kernel construct sizes clauses
Loop construct	Multiple device targets	Interface for profiling and tracing
Data construct	Atomic directive	
Parallel construct		
Kernels construct		

OpenMP should just merge, which is discussed below in the context of developing linear algebra libraries.

Compiler Support

The most mature OpenACC compilers are available as commercial software. PGI, Cray, and CAPS are leading the path for proprietary OpenACC compilers. The PGI Accelerator Compilers version 2017 targets NVIDIA GPUs, with a support of the OpenACC 2.5 standard. The collection includes C, C++, and Fortran compilers. The Cray Compilation Environment (CCE) also supports OpenACC on Cray systems. CCE 8.2 supports OpenACC 2.0.

Development of open source OpenACC compilers is underway. The GNU Compiler GCC 6 release series provides a partial implementation of the OpenACC 2.0a specification. It works only for NVIDIA PTX targets. OpenARC is an open source compiler developed at ORNL. It provides a full support for the v1.0 specifications and a subset of the 2.0 specifications. Other open source projects include RoseACC (University of Delaware and LLNL), OpenUH (University of Houston), and the Omni compiler (RIKEN AICS/University of Tsukuba).

OpenACC vs. OpenMP

OpenMP has been around for longer, has much more features, and as related to accelerators, is also catching up by taking advantage of the innovations driven by OpenACC and the developers/community that they share. Still, there are some significant differences between them. In particular, OpenACC continues to be more mature for scalable computing with accelerators, vs. OpenMP for general purpose parallelism for multicore. A notable difference that is often highlighted is also that OpenACC is more *descriptive*, vs. OpenMP being more *prescriptive*. In other words, OpenACC gives more freedom to just annotate where parallelism is, and thus leaves the rest to the compiler, vs. OpenMP provides more features to be directed by the programmer to specify detail on the implementation. This has been noted to leave OpenMP less performance portable across different architectures, as tuning parallelism for one architecture does not guarantee efficiency for another.

However, this difference is not that critical for the development of high-performance linear

Table 4.2: High level comparison of the latest released standards of OpenACC and OpenMP.

OpenACC 2.5	OpenMP 4.0
No goal of general purpose	Focused on general purpose parallelism
Focused on accelerator hardware	Focused on multicore, acceleration optional
Performance portability possible	Performance portability requires effort
Descriptive (functional)	Prescriptive (procedural)
Interoperability available	Interoperability still evolving

algebra libraries, where developers can afford to concentrate on individual routines for particular architectures. Typically, the goal is to still have a single source, but that source is highly parameterized, so that the best-performing version can be automatically discovered through empirically-based autotuning (vs. reliance on a compiler). Furthermore, high-level libraries, like LAPACK, rely on a number of highly tuned kernels, e.g., BLAS, that are also not derived as a single source, but rather, assembly implementations of algorithms that are themselves specifically designed and tuned for target architectures.

Advantages and Disadvantages

OpenACC main advantages can be summarized as follows:

- Scientists and application programmers can quickly determine if their codes will benefit from acceleration. There is no need to develop low-level kernels to reach such a decision.
- Since it is a directive based approach, it requires minimal structural code changes, compared to lower-level languages such as CUDA and OpenCL.
- Software portability: the same code base can run on many platforms, with and without accelerators. It can even be compiled using compilers with no OpenACC support, since the directives will be ignored.
- Performance portability: the task of discovering parallelism and mapping it to the hardware is left to the compiler. The programmer just "hints" the compiler about potential parallel regions. In general, the more hints, the more performance is obtained.
- Cross platform: for example, the same code base runs on NVIDIA and AMD GPUs. CAPS even provides support for OpenACC on the Intel Xeon Phi coprocessor.
- Interoperability: the programmer can choose to accelerate parts of the code using directives, and other parts using calls to accelerated libraries (e.g. cuBLAS).

On the other hand, here are some downsides for development using OpenACC:

- Explicit management of data movement: In the case of separate memory spaces for the host and the accelerator, the default assumption in OpenACC is that the data is in the host memory space (both inputs and outputs). Without explicit user control, the compiler will create memory allocations and data copies as needed at every parallel region. The resulting code, therefore, might be slower than the non-accelerated code.

- OpenACC is not yet fully adopted by major vendors like Intel and AMD.
- Compilers that fully support OpenACC are proprietary. No mature open source alternatives are available yet.
- OpenACC accelerated codes are usually outperformed by those written in the lower-level language (e.g. CUDA). The portability of OpenACC comes at the cost of the inability of take advantage of some architecture specific features. However, OpenACC provides some advanced optional controls through which programmers can improve the performance of their codes.

The OpenACC Accelerator Model

In order to ensure portability to multiple computing architectures, OpenACC defines an abstract model for accelerated computing. This model exposes multiple levels of parallelism that may appear in a processor, as well as a hierarchy of memories with varying degrees of speed and addressability. The model ensures that OpenACC is applicable to different, current and future, architectures. At its core, OpenACC supports offloading of both computation and data from a *host device* to an *accelerator device*. These devices may be the same or may be completely different architectures. Such is the case of a CPU+GPU configuration. The two devices may also have separate memory spaces or a single memory space. In the case that the two devices have different memories, the OpenACC compiler and runtime will analyze the code and handle any accelerator memory management and the transfer of data between the host and the accelerator memories. Figure 4.6 shows a high level diagram of the OpenACC abstract accelerator model.

OpenACC programmers should think of variables as objects, regardless of their locations. This is different from the common way of associating the variable name with the memory space where it resides. For example, in a CPU+GPU configurations, that are programmed using CUDA or OpenCL, a single variable `A` is expanded to variables that are actually copies (e.g., `h_A` in the host memory and `d_A` in the accelerator memory). In the OpenACC model it is preferred to decouple a variable from its location in order to avoid portability issues on systems with a shared memory space between the host and the accelerator.

There are three levels of parallelism in OpenACC: *gang*, *worker*, and *vector*. There is also a fourth level (*seq*) that indicates that a code segment should not be parallelized. The *vector* level has the finest granularity, while the *gang* level has the coarsest one. The *worker* level is a medium level between the former two. A gang consists of one or more workers. A worker operates on a vector of a certain length. Gangs are totally independent from each other, and cannot synchronize. On the other hand, workers and vectors inside the same gang can synchronize and share data through a fast memory level (e.g. a cache or shared memory). Vector execution is similar to SIMD parallelsim, with a single instruction being executed on multiple pieces of data (vector length). The OpenACC model exposes a cache memory within each gang, which is shared by all workers and vectors of that gang. Figure 4.7 visualizes the OpenACC levels of parallelism.

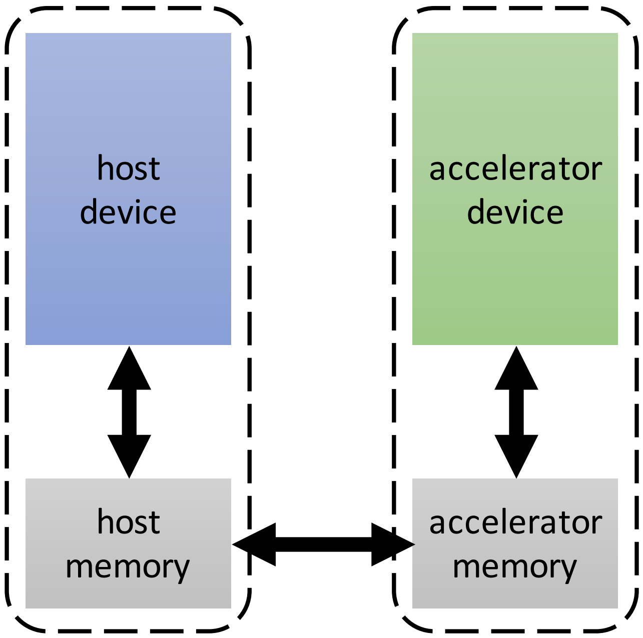


Figure 4.6: OpenACC abstract accelerator model.

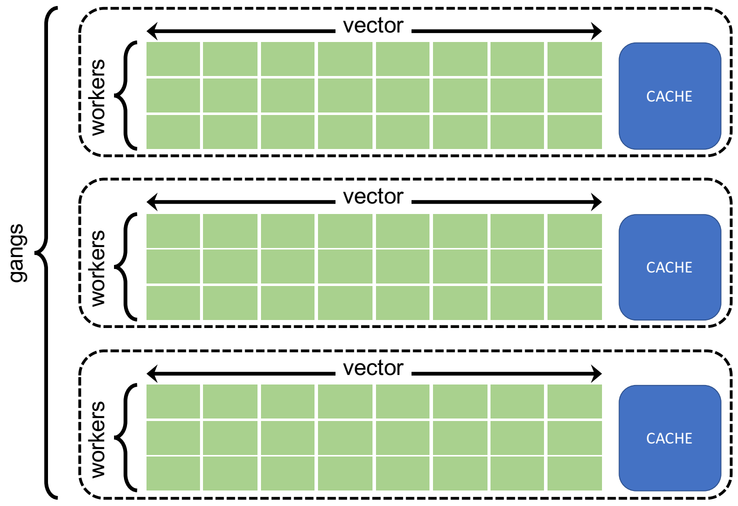


Figure 4.7: OpenACC levels of parallelism.

It should be noted that OpenACC programmers do not need to control these levels explicitly. The default mode of operation is that the compilers maps parallel regions of the code automatically into these levels according to its knowledge about the target device. However, a programmer can control these levels of parallelism for the sake of having more control, or further tuning for a specific hardware. The explicit programmer control comes, however, at the cost of less portability for other devices.

Recommended Porting Procedure to OpenACC

As the best practice, here are four main steps to accelerate an application using OpenACC:

1. Identify the most time consuming loops and blocks of the code. This can be done using performance profiling tools.
2. Decorate parallel loops within the code with OpenACC directives that provide the information necessary to parallelize the code for the target architecture.
3. Use OpenACC directives to optimize data locality and eliminate unnecessary copies between the host and the accelerator. Since OpenACC targets systems with both the same memory and separate memories, OpenACC compilers are conservative with respect to data locality. Unless otherwise specified by the user, the compiler will handle accelerated regions as standalone transactions with the accelerator. Each transaction consists of the copy-in, compute, and copy-out stages. Obviously, this model is far from optimized, due to the redundancy in data movement. The programmer can provide information to the compiler to keep the data as long as possible in the accelerator memory, before it is copied back to the host, thus maximizing data locality.
4. Further loop-level optimization: compilers discover and map the parallelism of a loop to the target hardware, based on internal heuristics and conservative rules. Additional performance gains can be obtained by providing more information to the compiler about the loop structure, and also tuning the offloaded code to the accelerator.

Example Code: Jacobi Iteration

This section introduces an example code in C/C++ that will be considered for acceleration using OpenACC. The code solves the 2D-Laplace equation with the iterative Jacobi solver. Iterative methods are a common technique to approximate the solution of elliptic PDEs, like the 2D-Laplace equation, within some acceptable tolerance. The code performs a simple stencil calculation, where the value for each point is calculated as the mean of its neighbors' values. The algorithm continues to iterate until either the maximum change in value, between two iterations, drops below some tolerance level or a maximum number of iterations is reached. The main iteration loop, written in C/C++, is shown in Figure 4.8.

The `while` loop is the *convergence loop*, since it contains a check for the stopping criteria (acceptable tolerance or maximum iterations). There are two properties of the convergence loop that are not parallel-friendly across iterations. The first is that the execution of an iteration is conditional based on the evaluation of the stopping criteria. The second is that computing the values of `A` depends on the values computed in the previous iteration, which imposes data dependencies among consecutive iterations.

However, the first loop nest has parallelization opportunities. This segment of the code loops over the 2D domain to compute the new values of `A`. By using an additional workspace `Anew`, all the computations of the new elements of `A` are independent, and can occur in parallel. The same loop nest has a *reduction operation* to find the maximum absolute error across the domain of `A`. Similarly, the second loop nest copies `Anew` back to `A`, which can also occur in parallel. A basic profiling experiment shows that, obviously, the two loop

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1] +
                                  A[j-1][i] + A[j+1][i]    );
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    if(iter % 100 == 0) printf("%5d,␣%0.6f\n", iter, error);
    iter++;
}
```

Figure 4.8: A sample Jacobi iteration. Error is printed every 100 iterations.

nests are the most time consuming parts of the whole solver. The next sections discuss the different OpenACC directives and how they can be applied to the Jacobi solver.

The `kernels`, `parallel`, and `loop` directives

These are the most common OpenACC directives for informing the compiler about code parallelization. The `kernels` directive identifies a region of code that may contain parallelism, but relies on the automatic parallelization capabilities of the compiler to analyze the region, identify which loops are safe to parallelize, and then accelerate these loops. Developers with little or no parallel programming experience, or those working on functions containing many loop nests that might be parallelized, will find the kernels directive a good starting place for OpenACC acceleration. Figure 4.9 demonstrates the use of `kernels` in C/C++.

```
#pragma acc kernels
{
    for (i=0; i<N; i++) {
        y[i] = 0.0f;
        x[i] = (float)(i+1);
    }

    for (i=0; i<N; i++) {
        y[i] = 2.0f * x[i] + y[i];
    }
}
```

Figure 4.9: An example for the `kernels` directive.

The `kernels` directive does not assert parallelization. In fact, the `kernels` directive gives the compiler the complete freedom to discover parallelism, analyze loops, and resolve data dependencies. The programmer just “tells” the compiler to search for parallelism opportunities in code enclosed between braces. The compiler will not parallelize loops or code segments unless it is certain about the safety to do so. In other words, a loop that is legitimately parallel can be ignored by the compiler, because the loop does not pass its conservative criteria for parallelization.

The `parallel` directive is more explicit, and is often associated with the `loop` directive. It hints the compiler that this loop or code block is safe to parallelize. The same example in Figure 4.9 can be parallelized using the combined `parallel loop` directive, as shown in Figure 4.10. Unlike the `kernels` directive, each loop needs to be explicitly decorated with the `parallel loop` directive. This is because the `parallel` directive relies on the programmer to identify the parallelism in the code rather than performing its own compiler analysis of the loops. In other words, the `kernels` directive may be thought of as a hint to the compiler of where it should look for parallelism, while the `parallel` directive is an assertion to the compiler of where there is parallelism. It is important to point out that only the availability of parallelism is defined, but the compiler still has the sole responsibility of mapping the parallelism to the target hardware, which is the ultimate requirement for portability.

```
#pragma acc parallel loop
for (i=0; i<N; i++) {
    y[i] = 0.0f;
    x[i] = (float)(i+1);
}

#pragma acc parallel loop
for (i=0; i<N; i++) {
    y[i] = 2.0f * x[i] + y[i];
}
```

Figure 4.10: An example for the combined `parallel loop` directive.

Another notable difference between the `kernels` and `parallel loop` directives is data movement. When using the `kernels` directive, any data copies occur at the beginning and the end of the decorated block of code, meaning that the data will remain on the device for the full extent of the region, or until it is needed again on the host within that region. This means that, if multiple loops access the same data, it will be copied to the accelerator once. When `parallel loop` is used for two subsequent loops that access the same data, the compiler may or may not copy the data back and forth between the host and the device between the two loops. For example, the compiler will generate an implicit data movement for each `parallel loop` in Figure 4.10, but it will generate data movement once for the `kernels` approach in Figure 4.9, which results in less communication by default.

The `loop` construct gives the compiler additional information about the very next loop in the source code. The `loop` directive was shown above in connection with the `parallel` directive, although it is also valid with the `kernels` directive. Loop clauses come in two forms: clauses for correctness and clauses for optimization. Optimization clauses are discussed later on. In order to maintain the correctness of the loop after parallelization, some clauses can be appended to the `loop` directive:

1. The `private` clause, which is used as `private(variable)`, specifies that each loop iteration requires its own copy of the listed variable(s). For example, if each loop contains a temporary array named `tmp`, that it uses during its calculation, then this variable is made private to each loop iteration in order to ensure correct results. If

`tmp` is not declared private, then threads executing different iterations may access this shared variable in unpredictable ways, resulting in race conditions and potentially incorrect results. Loop iterators are private by default. In addition, and unless otherwise specified, any scalar variable accessed within a parallel loop is made *first private* by default, meaning that a private copy is made of the variable for each loop iteration and it is initialized with the value of that scalar upon entering the region. Finally, any variables that are declared within a loop in C or C++ is made private to the iterations of that loop by default.

2. The `reduction` clause, which is written as `reduction(operator:variable)`, works similarly to the `private` clause in that a private copy of the affected variable is generated for each loop iteration, but `reduction` goes a step further to reduce all of those private copies into one final result, which is returned from the region. For example, the maximum of all private copies of the variable may be required or perhaps the sum. A reduction may only be specified on a scalar variable and only common specified operations can be performed, such as $+$, $*$, `min`, `max`, and many bitwise operations.

Parallelizing Jacobi Iteration using OpenACC

The example code shown in Figure 4.8 is dominated by the two loop nests, which can be parallelized in OpenACC either using the `parallel loop` or the `kernels` directives. This section shows both approaches.

Figure 4.11 shows the parallelization using the `parallel loop` directive, where each loop nest is annotated with the directive. Some compilers will also analyze the innermost loops and determine that it is also safely parallel. However, it is a better practice for portability to explicitly inform the compiler about safely parallelizable loops. The directive on top of the first loop nest also informs the compiler about the `max` reduction operation required on the variable `error`. The innermost loops can be also annotated with `parallel loop`, although most compilers will parallelize them using the `parallel` directive only.

Figure 4.12 shows the parallelization using the `kernels` directive. Note that the two loop nests are combined into one parallel region that is annotated using the `kernels` directive. This means that the programmer grants the compiler the freedom to analyze and discover parallelism. For this relatively simple example, most compilers will be able to discover that all loops are safely parallel.

The previous code examples discussed how to map parallelism to the accelerator, but dropped the discussion about data movements between the host and the accelerator. This is why the code examples of Figures 4.11 and 4.12 are unlikely to produce performance gains against non-accelerated codes. In the absence of user-defined controls for communication, the OpenACC compilers adopt a conservative strategy, with respect to data movements, by copying the data back and forth between the host ans the accelerator at the beginning and end of every parallel region. In this regard, the accelerated codes of Figures 4.11 and 4.12 will be dominated by data copies and the runtime overhead to setup such copies, with the code shown in Figure 4.12 being faster, due to having just one parallel region instead of two. This is why the user has to provide more information to the compiler regarding data locality on the accelerator side.

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;

    #pragma acc parallel loop reduction(max:error)
    for( int j = 1; j < n-1; j++) {
        #pragma acc loop reduction(max:error)
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = 0.25 * ( Anew[j][i+1] + Anew[j][i-1] +
                               Anew[j-1][i] + Anew[j+1][i]   );
            error = fmax( error, fabs(A[j][i] - Anew[j][i]));
        }
    }

    #pragma acc parallel loop
    for( int j = 1; j < n-1; j++) {
        #pragma acc loop
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    if(iter % 100 == 0) printf("%5d,␣%0.6f\n", iter, error);
    iter++;
}
```

Figure 4.11: Jacobi iteration using the `parallel loop` directive.

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;

    #pragma acc kernels
    {
        for( int j = 1; j < n-1; j++) {
            for( int i = 1; i < m-1; i++ ) {
                A[j][i] = 0.25 * ( Anew[j][i+1] + Anew[j][i-1] +
                                   Anew[j-1][i] + Anew[j+1][i]   );
                error = fmax( error, fabs(A[j][i] - Anew[j][i]));
            }
        }

        for( int j = 1; j < n-1; j++) {
            for( int i = 1; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
            }
        }
    }

    if(iter % 100 == 0) printf("%5d,␣%0.6f\n", iter, error);
    iter++;
}
```

Figure 4.12: Jacobi iteration using the `kernels` directive.

Improving Data Locality

This section discusses the main user-defined controls for improving data locality on the accelerator. This is a must-do optimization when the host and the accelerator have two separate memories.

A *data region*, which is defined using the `data` directive, is used to share data across multiple parallel regions that are defined in the same scope. It can also be placed at a higher level in the program call tree to enable data sharing among parallel regions in multiple functions. The `data` directive can be used in the code example shown in Figure 4.11, in order to eliminate unnecessary data copies between the two loop nests. Figure 4.13 shows another example, where the data region enables the `x` and `y` arrays to be reused between the two parallel regions. This removes any data copies that happen between the two regions, but it still does not guarantee optimal data movement. In order to provide the information necessary to perform optimal data movement, the programmer can add data clauses to the data region. Note that an implicit data region is created by each `parallel` and `kernels` region.

```
#pragma acc data
{
    #pragma acc parallel loop
    for (i=0; i<N; i++) {
        y[i] = 0.0f;
        x[i] = (float)(i+1);
    }

    #pragma acc parallel loop
    for (i=0; i<N; i++) {
        y[i] = 2.0f * x[i] + y[i];
    }
}
```

Figure 4.13: An example of a data region enclosing two parallel loops.

OpenACC supports a number of *data clauses* that enable the programmer to have more control over data copies, allocations, and deallocations. Table 4.3 summarizes such clauses.

The OpenACC 1.0 and 2.0 standards also have `present_or_*` clauses (e.g. `present_or_copy`). Such clauses inform the compiler to use the present copy of the listed variable if it exists. If it does not, then the compiler performs the normal action of the clause, as described in Table 4.3. These clauses are frequently abbreviated, like `pcopyin` instead of `present_or_copyin`. OpenACC 2.5 modifies the behavior of the mentioned clauses so that they all test the presence by default (e.g. `copy` becomes equivalent to `present_or_copy`).

Information about array sizes can be also passed to OpenACC compilers. This is particularly important for C/C++ compilers, which cannot implicitly deduce the size of the array to be allocated or copied. The syntax of specifying array information takes the form `x[start:count]`, where `start` is the first element and `count` is the number of element. The same syntax works for allocations and copies of arrays. Figure 4.14 shows an example code that uses data clauses, and passes array information through OpenACC directives. The example code has two safely parallel `for` loops. The input array `x` has `N` elements, and is input only. Using the `pcreat` clause, the compiler allocates `x` only if it is not present on the accelerator memory. A similar behavior occurs with `y`, except that the array is output only. Without the use of data clauses, the compiler will perform unnecessary data copies, like copying `x` and `y` at the beginning of the parallel region, which is not needed since the first loop already sets them. Another unnecessary copy that is avoided is copying back `x` to the host, since it is an input only array.

Having discussed the different `data` clauses in OpenACC, it is time to show a more optimized version of the Jacobi iteration case study that uses data clauses to improve locality on the

Data clause	Description
`copy`	Create space for the listed variables on the device, initialize the variable by copying data to the device at the beginning of the region, copy the results back to the host at the end of the region, and finally release the space on the device when done.
`copyin`	Create space for the listed variables on the device, initialize the variable by copying data to the device at the beginning of the region, and release the space on the device when done without copying the data back the the host.
`copyout`	Create space for the listed variables on the device but do not initialize them. At the end of the region, copy the results back to the host and release the space on the device.
`create`	Create space for the listed variables and release it at the end of the region, but do not copy to or from the device.
`present`	The listed variables are already present on the device, so no further action needs to be taken. This is most frequently used when a data region exists in a higher-level routine.
`deviceptr`	The listed variables use device memory that has been managed outside of OpenACC, therefore the variables should be used on the device without any address translation. This clause is generally used when OpenACC is mixed with another programming model.

Table 4.3: OpenACC data clauses.

```
#pragma acc data pcreate(x[0:N]) pcopyout(y[0:N])
{
    #pragma acc parallel loop
    for (i=0; i<N; i++) {
        y[i] = 0.0f;
        x[i] = (float)(i+1);
    }

    #pragma acc parallel loop
    for (i=0; i<N; i++) {
        y[i] = 2.0f * x[i] + y[i];
    }
}
```

Figure 4.14: An example code that uses data clauses with array information.

accelerator side. Figure 4.15 shows an example that uses `parallel loop` for parallelization, and the data clauses `copy` and `create` for data movement. Note that the `A` array is copied to the accelerator at the beginning of the parallel region, and then back to the host at the end of it. The array `Anew` is used internally only, more like a workspace. Therefore, it is created on the accelerator upon entry to the parallel region, with no copies required between the host and the device. The data clauses for `A`/`Anew` use the array information to specify the copy/allocation size, respectively.

```
#pragma acc data copy(A[1:n][1:m]) create(Anew[n][m])
while ( error > tol && iter < iter_max ) {
    error = 0.0;

    #pragma acc parallel loop reduction(max:error)
    for( int j = 1; j < n-1; j++) {
        #pragma acc loop reduction(max:error)
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1] +
                                  A[j-1][i] + A[j+1][i]   );

            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma acc parallel loop
    for( int j = 1; j < n-1; j++) {
        #pragma acc loop
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    if(iter % 100 == 0) printf("%5d,␣%0.6f\n", iter, error);
    iter++;
}
```

Figure 4.15: Improving data locality for a parallelized Jacobi iteration.

Advanced Programming using OpenACC

This section discusses some advanced features of OpenACC. These features give the programmer more controls that are not available through the basic directives for parallelization and data movements.

Unstructured Data Scopes: The `data` directive discussed above assumes that the data are allocated and deallocated in the same scope. In many cases, structured data lifetime is not applicable. For example, assume a C++ class where the data is created using a constructor, and then freed in a destructor. This is clearly a situation that cannot be resolved using the `data` directive. Since OpenACC 2.0, it is possible to have unstructured data scopes using the `enter data` and `exit data` directives.
The `enter data` directive accepts the `create` and `copyin` data clauses and may be used to specify when data should be created on the device. The `exit data` directive accepts the `copyout` and a special `delete` data clause to specify when data should be removed from the device. Please note that multiple `enter data` directives may place an array on the device, but when any `exit data` directive removes it from the device, it will be immediately removed, regardless of how many `enter data` regions reference it. Figure 4.16 shows a simple C++ class example that has a constructor, a destructor, and a copy constructor. Note that the constructor copies the `this` pointer to the accelerator as well, in order to ensure that the scalar member `len` and the pointer `arr` are available on the accelerator as well as the host. The copy constructor uses a `parallel loop` to perform the copy from an object that is resident in the accelerator memory, hence the use of the `present` clause.

The `update` Directive: The update directive provides a way to explicitly update the values

```
template <class ctype> class Data
{
    private:
    int len;    // length of the data array
    ctype *arr; // data array

    public:
    // class constructor
    Data(int length) {
        len = length;
        arr = new ctype[len];
        #pragma acc enter data copyin(this)
        #pragma acc enter data create(arr[0:len])
    }

    // copy constructor
    Data(const Data<ctype> &d)
    {
        len = d.len;
        arr = new ctype[len];
        #pragma acc enter data copyin(this)
        #pragma acc enter data create(arr[0:len])
        #pragma acc parallel loop present(arr[0:len],d)
        for(int i = 0; i < len; i++){
            arr[i] = d.arr[i];
        }
    }

    // class destructor
    ~Data() {
        #pragma acc exit data delete(arr)
        #pragma acc exit data delete(this)
        delete arr;
        len = 0;
    }
};
```

Figure 4.16: An example of unstructured data scopes.

of host or device memory with the values of the other. This can be thought of as synchronizing the contents of the two memories. As of OpenACC 2.0, the `update` directive accepts a `device` clause for copying data from the host to the device and a `self` clause for updating from the device to local memory, which is the host memory, except in the case of nested OpenACC regions. Figure 4.17 shows an example of the `update` directive that may be added to the C++ class in Figure 4.16.

```
void update_host() {
    #pragma acc update self(arr[0:len])
    ;
}

void update_device() {
    #pragma acc update device(arr[0:len])
    ;
}
```

Figure 4.17: An example of the `update` directive.

Loop Optimization: A programmer may choose to further optimize a loop by explicitly mapping the parallelism to gangs, workers, and vectors. In fact, the `loop` directive can be combined with the following clauses:

1. A `gang` clause, which partitions the loop across gangs.
2. A `worker` clause, which partitions the loop across workers.
3. A `vector` clause, which vectorizes the loop.
4. A `seq` clause, which runs the loop sequentially.

These directives may also be combined for a particular loop. For example, a gang vector loop would be partitioned across gangs, each of which with one worker implicitly, and then vectorized. The OpenACC specification enforces that the outermost loop must be a gang loop, the innermost parallel loop must be a vector loop, and a worker loop may appear in between. A sequential loop may appear at any level. The programmer can also control the number of gangs, workers, and vectors used in partitioning the loop.

Routine Parallelism: If a function is called within a parallel loop, the compiler might not be able to parallelize it correctly, since it has no information about the loop structure of that function. OpenACC 1.0 either inlines all function calls in the parallel region, or decides not to parallelize the region at all. The OpenACC 2.0 introduced a new directive `routine` that is used to inform the compiler about potential parallelism in a certain routine. The `routine` must be added to the function definition.

Asynchronous operations: It is possible to perform operations asynchronously using OpenACC. The `async` clause allows a parallel region to be executed on the accelerator without the host waiting for it to finish. The `async` clause can be used with the `kernels`, `parallel loop`, and `update` directives. In order to synchronize the host and the accelerator back, a `wait` directive can be used. Both `async` and `wait` accept a non-negative integer value which indicates the queue id to execute in or to synchronize against.If no queue is passed, the execution/synchronization occurs with respect to the default queue.
Figure 4.18 shows an example that uses asynchronous operations. The code initializes the arrays `a` and `b` using different queues, so that they can be done concurrently. The `wait(1) async(2)` statement makes all future launches in queue 2 dependent on the completion of all tasks submitted to queue 1. The vector addition can be then safely submitted to queue 2. the The code then updates the copy of `c` on the host on queue 2. Finally, the `wait` statement ensures that the host waits for all previous operations to complete.

Multi-Device Acceleration: OpenACC supports multi-accelerator programming using a set of APIs that can read the number of devices in a system, select a particular accelerator, and get the ID of the currently selected accelerator. In the case of having different accelerator types, there are APIs that can query or set the type of a particular accelerator in the system.

OpenACC Interoperability

OpenACC codes can be mixed with codes that use other programming models, such as CUDA and OpenCL. In such case, the programmer should decide between either managing

```
#pragma acc parallel loop async(1)
for (int i=0; i<N; i++){
    a[i] = i;
}

#pragma acc parallel loop async(2)
for (int i=0; i<N; i++) {
    b[i] = 2*i;
}

#pragma acc wait(1) async(2)
#pragma acc parallel loop async(2)
for (int i=0; i<N; i++) {
 c[i] = a[i] + b[i]
}

#pragma acc update self(c[0:N]) async(2)
#pragma acc wait
```

Figure 4.18: Asynchronous operations in OpenACC.

the device memory inside the context of OpenACC or in the context of other programming models. For example, there should be a way to pass device arrays created inside OpenACC to other CUDA libraries (e.g. cuBLAS), and vice versa. In general there are two ways to accomplish such interoperability.

1. The first is *host data region*, which is used when the device variables are created and managed inside OpenACC. In such a case, the host can call other accelerated libraries by extracting the device pointers from OpenACC using the `host_data` region. The `host_data` region gives the programmer a way to expose the device address of a given array to the host for passing into a function. This data must have already been moved to the device previously. The region accepts only the `use_device` clause, which specifies which device variables should be exposed to the host. Figure 4.19 shows an example for two arrays, `x` and `y`, which are placed on the device using a `data` region and then initialized in an OpenACC loop. These arrays are then passed to the `cublasSaxpy` function as device pointers using the `host_data` region.
2. The second way is using *device pointers*. In this case, the device variables are created and managed outside the OpenACC context. In order to pass such variables to OpenACC regions, the `device_ptr` data clause must be used. Figure 4.20 shows an example for coding an OpenACC equivalent to `cublasSaxpy`.

4.1.4 MPI

Message Passing Interface (MPI) defines a standard interfaces for cross-platform programming on distributed-memory computers. It includes a wide range of functions that allow the processes to pass messages in a portable fashion. It is designed by a group of experts from both acadmia and industry, and is defined for both C and Fortran. The first release of MPI, MPI-1.0, was in February 1993.

```
#pragma acc data create(x[0:n]) copyout(y[0:n])
{
    #pragma acc kernels
    {
        for( i = 0; i < n; i++) {
            x[i] = 1.0f;
            y[i] = 0.0f;
        }
    }

    #pragma acc host_data use_device(x,y)
    {
        cublasSaxpy(n, 2.0, x, 1, y, 1);
    }
}
```

Figure 4.19: OpenACC interoperability using `host_data` regions.

```
void saxpy(int n, float a, float * restrict x, float * restrict y) {
    #pragma acc kernels deviceptr(x,y)
    {
        for(int i=0; i<n; i++) {
            y[i] += a*x[i];
        }
    }
}
```

Figure 4.20: OpenACC interoperability using `device_ptr` data clause.

MPI-3

The most recent release of MPI is MPI-3.0 that was approved by the MPI Forum in September 2012, followed by MPI-3.1 in June 2015. The MPI-3.1 mostly consists of corrections and clarifications (e.g., for Fortran bindings), but also includes a few enhancements and new features (e.g., for portablity and nonblocking I/O). There are several implementations of MPI-3, including three open source implementations, MPICH and OpenMPI, and MVAPICH which is based on MPICH but adds a few features such as Infinitiband supports. Here we describe the main new features of MPI-3.

Nonblocking Collectives: A set of non-blocking, or "immediate," collectives are definted (e.g., `MPI_Ibcast`, `MPI_Ireduce`, `MPI_Iallreduce`, and `MPI_Ibarrier`, and `MPI_Test` or `MPI_Wait` to test or wait for the completion of the non-blocking collectives). Multiple nonblocking collectives may be pipelined by calling multiple non-blocking collective functions before calling the corresponding waits, though they must be called in the same order by all the processes. They allow the communication to overlap with computation or with other communications. They may be also used to mitigate load imbalance or system noise by avoiding immediate global synchronizations.

```
MPI_Ibcast(buf, count, type, root, comm, &request);
// do computation or communication
...
MPI_Wait(&request, &status);
```

In practice, its effectiveness to overlap the communication with the computation depends on many factors, including their implementations, the target hardware, and the nature of communications and computations. The current releases of both MPICH and OpenMPI implement the non-blocking collectives using TCP and IP-over-Infiniband, while in OpenMPI, the blocking collectives may directly take advantage of Inifinitiband. Hence, though the non-blocking communication may allow the communication to overlap, the actual communication could be slower than the corresponding blocking communication. To ensure the progress of the communication, both MPICH and OpenMPI provide an option to enable a progress thread. If the progress thread and application threads use the same resource, these threads may compete for the resource and slows down the execution of the application or the communication. In order to reduce the interference of the MPI's progress thread with the application threads, each process may require a separate spare core, or a hyper thread. Both MPICH and OpenMPI are actively working to improve the performance of the non-blocking collective (e.g., under the ECP OMPI-X project).

Neighborhood collectives: The collective communications among a subset of processes, named "neighbors," are defined, e.g.:

- `MPI_Neighbor_allgather`,
- `MPI_Neighbor_alltoall`,
- `MPI_Ineighbor_allgather`,
- `MPI_Ineighbor_alltoall`.

Such neighbors can be defined based on an MPI Cartesian or virtual process topology (MPI-2 introduced the creation of graph topology where each process specifies its neighbors).

```
// create a 2x2x2 3D periodic process grid
MPI_Cart_create(comm, 3, {2, 2, 2}, {1, 1, 1}, 1, &newcomm);
// start communication with neighbors
MPI_Ineighbor_alltoall(..., &newcomm, &req);
// do local computation
...
MPI_Wait(&req, MPI_STATUS_IGNORE);
// do boundary computation
...
```

Improved One-Sided Communication Interface: MPI-2 introduced one-sided communication between "origin" and "target" processes (e.g., `MPI_Put` and `MPI_Get`). This allows an origin process to access remote memory without synchronizing with the target process. Each process can assign its local memory as remotely accessible memory, which is called "window object" (using `MPI_Win_create`). There are three data access models, referred to as "active," "generalized active," and "passive" targets. For instance, for the active data access control, the process can create a "fence" and define an "epoch", within which the window object becomes available for all the processes. For the general active control, the process can specify which process it communicates with (e.g., using `MPI_Win_post` and `MPI_Win_start` with `MPI_Group`), while the passive control initiates the access to a specific rank.

```
// create window
MPI_Win window;
MPI_Win_create(local_memory, size, disp_unit, info, comm, &window);
// start of fence
MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), window);
// one-sided communication
```

```
MPI_Get(origin_addr, origin_count, origin_type,
        target_rank, target_distp, target_count, target_type, window);
MPI_Win_fence(MPI_MODE_NOSUCCEED, window);
// end of fence
MPI_Win_free(&window);
```

MPI-3 introduced new window creation routines (i.e., `MPI_Win_allocate`, `MPI_Win_create_dynamic`, or `MPI_Win_allocate_shared`) and atomic operations (i.e., `MPI_Get_accumulation`, `MPI_Fetch_and_op`, and `MPI_Compare_and_swap`). MPI-3 also allows "unified memory model" if available.

Fortran 2008 Bindings: MPI-3 now complies with the Fortran standard (with the `mpi_f08` module).

```
use mpi_f08
double precision, asynchronous :: buf(1000)
type(MPI_STATUS) :: status
type(MPI_Request) :: req
type(MPI_Comm) :: comm

call MPI_IRECV(buf, 1000, MPI_DOUBLE_PRECISION, 0, 0, comm, req)
... // computation or communication
call MPI_WAIT(req, status)

if (status%MPI_ERROR .eq. MPI_SUCCESS) then
    if (.not. MPI_ASYNC_PROTECTS_NONBLOCKING) then
        call MPI_F_SYNC_REF(buf);
    end if
end if
```

Tools Interface: MPI-3 introduced an interface for tools, called `MPI_T`. The interface provides mechanisms to access the control and performance variables exposed by MPI. It is complimentary to PMPI, and allows the performance profiler to extract information about the MPI processes (e.g., number of packets sent, time spent blocking, buffer memory, etc.). Several control variables are available including:

- `ALLTOALL_SHORT_MSG_SIZE`,
- `ALLTOALL_THROTTLE`,
- `BCAST_MIN_PROCS`, etc.

The code below shows an example of changing a control variable (i.e., doubling the short message size for alltoall):

```
MPI_T_init_thread(MPI_THREAD_SINGLE, &provided);
MPI_Init(&argc, &argv);
MPI_T_cvar_get_num(&cvar_num);
for (cvar_id = 0; cvar_id < cvar_num; cvar_id++) {
    MPI_T_cvar_get_info(cvar_id, name, &name_len, &verbosity, &dtype, &enumtype,
                        desc, &desc_len, &bind, &scope);
    if (strncmp(name, "ALLTOALL_SHORT_MSG_SIZE", STR_SZ) == 0) {
        // double the message size
        MPI_T_cvar_handle_alloc(cvar_id, NULL, &handle, &count);
        MPI_T_cvar_read(handle, &msg_size);
        msg_size *= 2;
        MPI_T_cvar_write(handle, &msg_size);
        MPI_T_cvar_handle_free(&handle);
        break;
    }
}
if (cvar_id == cvar_num) {
    printf("Error:␣ALLTOALL_SHORT_MSG_SIZE␣not␣available\n");
}
```

```
// do computation and communication including alltoall
...
MPI_Finalize();
MPI_T_finalize();
```

Available performance variables include:

- unexpected_recvq_length,
- unexpected_recvq_match_attempts,
- unexpected_recvq_buffer_size,
- mem_allocated,
- mv2_progress_poll_count, etc.

The code below shows an example of reading a performance variable:

```
MPI_T_pvar_get_num(&pvar_num);
for (pvar_id = 0; pvar_id < pvar_num; par_id++) {
    MPI_T_pvar_get_info(pvar_id, pvar_name, &name_len, &verbosity,
                        &variable_class, &data_type, &enum_type,
                        description, &description_len, &binding,
                        &readonly, &continuous, &atomic);
    if (strcmp(pvar_name, "mv2_progress_poll_count") == 0) {
        MPI_T_pvar_session_create(&pvar_session);
        MPI_T_pvar_handle_alloc(pvar_session, pvar_id, NULL,
                                &progress_poll_count_handle, &count);
    }
}

MPI_Isend(buf, buf_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &req);
MPI_T_pvar_read(pvar_session, progress_poll_count_handle, &poll_count);
printf("progress␣poll␣count␣=␣%d\n", poll_count);

MPI_T_pvar_handle_free(pvar_session, &progress_poll_count_handle);
MPI_T_pvar_session_free(&pvar_session);
```

Besides these, MPI-3 includes thread-safe probe and receive, noncollective communicator creation, and nonblocking communicator duplication.

MPI Features of Interests to SLATE

Thread Support: There are four levels of thread support that an MPI imlementation can provide:

- MPI_THREAD_SINGLE: Application is single-threaded.
- MPI_THREAD_FUNNELED: Application may be multi-threaded, but only the main thread makes the MPI calls.
- MPI_THREAD_SERIALIZED: Application is multi-threaded and any thread may make MPI calls. However, only one thread will call the MPI function at a time.
- MPI_THREAD_MULTIPLE: Application is multi-threaded, and any thread may make an MPI call at any time.

An application may request a certain level of thread support using MPI_Init_thread. Then, the MPI implementation informs the application of the highest level of thread suppor that it can provide.

One use of the MPI's multi-thread support could be to overlap the communication with the computation, or to pipeline different communications. Although non-blocking communication provides the potential to overlap or pipeline communication, without a progress thread, the non-blocking communication may progress only when the application thread is blocked in an MPI call (i.e., `MPI_Wait`). This could prevent the application from overlapping or pipelining the communication. Although the MPI's progress thread (with a spare core or hyper thread) may ensure the collective to progress in the background, the current implementation of the non-blocking collective may be based on TCP/IP while the corresponding blocking collective may be able to directly supports Infiniband. Hence, the non-blocking communication may be slower than the blocking communication. Alternatively, using MPI's thread support, if one of the application threads blocks on the communication, while the other threads perform the computation, or other communication, the application may achieve better overlap (depending on the nature of the communication). However, even when MPI supports multiple threads, its use from different threads must be carefully designed. For instance, if there are multiple outstanding all-reduce requests from different threads, the application must ensure that the requests are matched correctly (e.g., using different communicators). In addition, some blocking communications (e.g., all-reduce) may not pipeline even when they are called from different threads at the same time. On the other hand, the nonblocking communications are under active developments, and the aforementioned shortcomings of the non-blocking communication may be addressed in the near-future releases.

GPU Support: CUDA-aware MPI detects if the message is in the host or device memory, and enables the data transfer between the GPUs through the same interfaces. For example, a simple code below transfers data in the device memory, but using CUDA-aware MPI, the explicit transfer of the data from the device to the host memories are not needed:

```
// copy message from device to host (not needed with CUDA-aware MPI)
cudaMemocpy(h_buffer, d_buffer, size*sizeof(double), cudaMemcpyDeviceToHost);

// start sending message through MPI
MPI_Isend(h_buffer, size, MPI_DOUBLE, 1, 100, MPI_COMM_WORLD, request);
```

The main advantage of the CUDA-aware MPI are:

- GPUDirect is utlized to avoid some memory copies between communication buffers (e.g., host buffer). CUDA 5.0 introduced the GPUDirect RDMA (Remote Direct Memory Access). With this feature, the data can be directly moved from the local device memory to the remote device memory as RDMA network messages. Figure 4.21 compares the inter-GPU communication with or without CUDA-aware MPI.
- The different steps of the message transfer are automatically pipelined (see Figure 4.22d). It also eases the use of non-blocking communication between the GPUs.

OpenMPI-1.7 introduced the CUDA-aware message-passing for all the send and receive APIs, and blocking collectives. CUDA-aware non-blocking collectives and one-sided communication is not supported. MVAPICH2 also supports the CUDA-aware

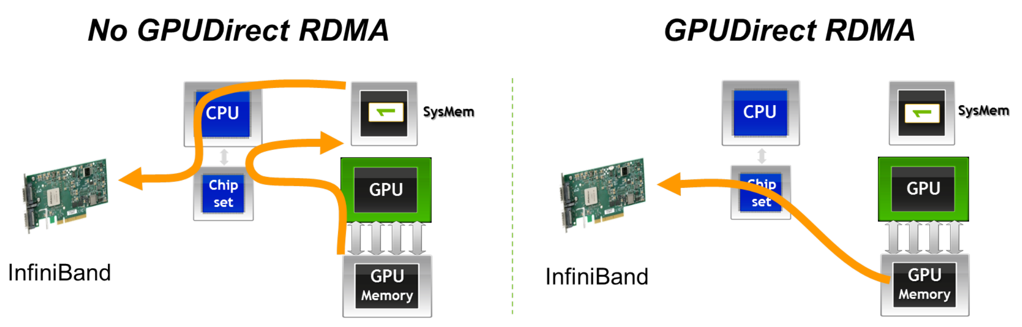
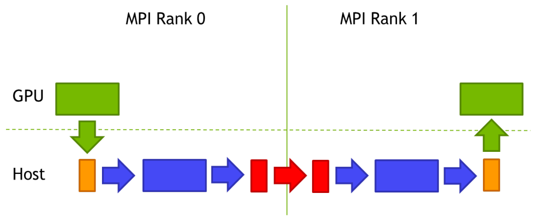
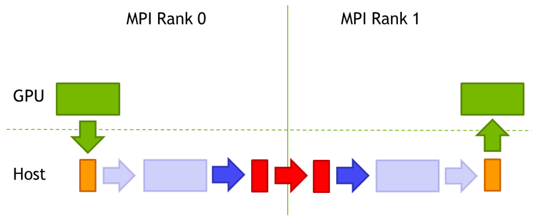


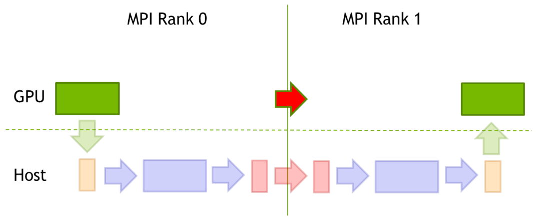
Figure 4.21: Illustration of GPUDirect (source: https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/).



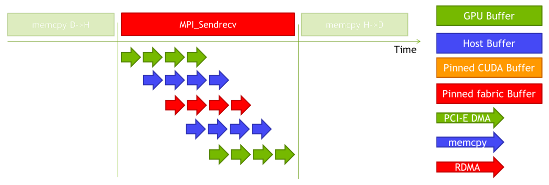
(a) Standard MPI.



(b) CUDA-aware MPI.



(c) CUDA-aware MPI with CUDADirect.



(d) Pipelining inter-GPU messaging.

Figure 4.22: Inter-node communication with CUDA-aware MPI with or without GPUDirect (source: https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/).

communication since 1.8 release. CUDA 4.0 is required while CUDA 4.1 adds IPC support for fast data transfer between the GPUs on the same node. CUDA 5.0 introduced GPUDirect RDMA.

MPI-4: Future Developments and Trends

MPI is a widely used library and, since its inception, has become ubiquitous in computational science and engineering. Future developments will address a broad spectrum of issues including:

- runtime interoperability for the MPI+X programming model,
- extending the MPI standard to better support the upcoming exascale architectures,

- improvements to scalability and performance,
- support for more dynamic execution environments,
- resilience,
- MPI tools interface.

The MPI standard, first published in 1994, was originally designed for architectures that were dramatically simpler than those of today, not to mention exascale. It has been evolved through the years to provide support for recent systems. However, changes in both the hardware and the software environments need to be studied in order to accommodate exascale computing. These key areas of concern have been identified by the Department of Energy, for the Exascale Computing Project.

MPI-4: Exascale Software Development Needs

Although current MPI implementations are well established, they do not provide everything that be needed to support exascale systems. Future developments will require out a broad spectrum of activities intended to address key concerns for exascale application.

- Since MPI typically now lives in a much more complex runtime environment than in the past, current research focuses on addressing the issues related to runtime interoperability for the MPI+X programming model. This includes issues like interactions with different models of multithreading and accelerator offload.
- In order to better support exascale machines, with massively threaded nodes and networks capable of very high bandwidths and message injection rates, recent efforts focus on issues such as: better scalability, resource management and isolation, petter performance of collective operations, optimization or MPI-3 RMA operations, etc. MPI developers will work closely with application teams and library writers to prototype, evaluate, and refine the standard and its implementations.
- As system environments grow more complex, there is a growing need to improve MPI support for more dynamic execution environments. This includes enhancing MPI implementations to facilitate intelligent decisions about process placement, as well as opening up new capabilities to respond to, and even coordinate with, other network users to manage contention. Memory hierarchies are also becoming deeper and more complex, and thus MPI developers must address data placement and movement in concert with other system components.
- Current work on error resilience of MPI is focused interoperability between the applications, the libraries, and the system software. Interfaces are needed for MPI to work effectively with sophisticated checkpoint/restart systems, which will be developed by the ECP. There is a need to develop the interfaces necessary to connect MPI to the supercomputerffs RAS subsystem, to allow more direct and more flexible error detection, in support of better fault handling capabilities.

- The availability of appropriate tools is critical to the ability of users to assess how well MPI is working for them. Future work on MPI tools interfaces will address some of the key emerging areas, including memory tracking, RMA performance data, access to network performance counters, etc.
- Finally, in order to deliver production-quality implementations, MPI developers will need to improve the testing infrastructure, and deploy extensive test suites.

4.2 Frameworks

4.2.1 PaRSEC

PaRSEC, short for Parallel Runtime Scheduling and Execution Controller, is a framework for managing the scheduling and execution of tasks on distributed many-core heterogeneous architectures [20, 21]. PaRSEC executes a workload defined as a Directed Acyclic Graph (DAG) consisting of the tasks (nodes) and their data-dependencies (edges). Figure 4.23 illustrates the PaRSEC framework. Internally the PaRSEC framework uses a Parameterized Task Graph (PTG) representation [26].

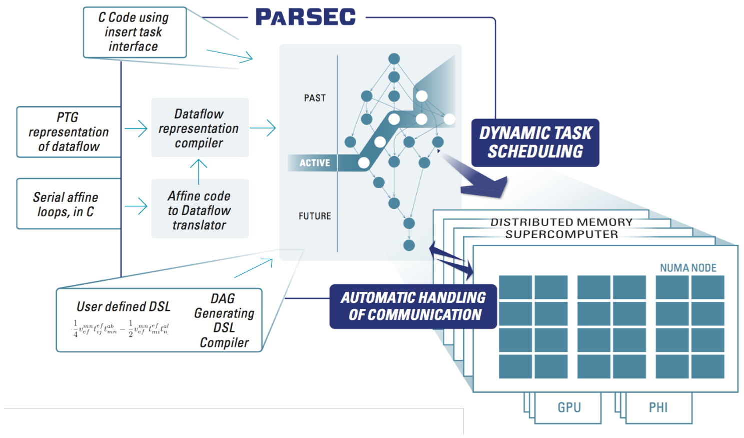

C Code using insert task interface
PARSEC
PTG representation of dataflow
Dataflow representation compiler
Serial affine loops, in C
Affine code to Dataflow translator
PAST
ACTIVE
FUTURE
DYNAMIC TASK SCHEDULING
DISTRIBUTED MEMORY SUPERCOMPUTER
NUMA NODE
User defined DSL
DAG Generating DSL Compiler
AUTOMATIC HANDLING OF COMMUNICATION
GPU
PHI

Figure 4.23: The PaRSEC framework.

It was first released in January 2012, followed by a couple of subreleases in 2014 (version 1.1 in January and 1.2 in April). It is now in the preparation for the 2.0 release. It has been used for solving dense linear algebra problems, and its releases are coupled with the releases of DPLASMA, the distributred memory counterpart of the PLASMA software. However, PaRSEC can be separately compiled on its own, and has been used for other purposes (e.g., sparse direct solver PaStiX and applications such as NWChem or TOTAL). Ongoing work includes supports for fault-tolerance, performance profiling, tracing and visualization.

Parameterized Task Graph

In the PTG, an algorithm is viewed as a DAG of tasks and the associated dataflow between the tasks. The computational work consists of task classes, parameterized by their position in the flow of computation, and the predecessor-successor relation between instances of the task classes is explicit in the task description, linking task instances by algebraic expressions of the task parameters. Figures 4.24 and 4.25 show the serial implementation of the QR factorization, and the PTG representation of the algorithm in the Job Description Format (JDF), respectively. In addition to the task-dependencies, the user must specify the initial data distribution, and the task is assigned to the process based on the data distribution. For instance, in Figure 4.25a, the `POTRF(k)` task is assigned to the process that owns the diagonal block `dataA(k, k)` as specified in Line 5.

```
for( int k = 0; k < n; k++) {
    potrf("Upper", A(k, k));

    for (int j = k+1; j < n; j++)
        trsm("Left", "Upper", "NoTrans", "NoTrans", A(k, k), A(k, j));

    for( int i = 0; i < n; i++ ) {
        syrk("Upper", "Trans", A(k, i), A(i, i));
        for (int j = i+1; j < n; j++) {
            gemm("Trans", "NoTrans", A(k, i), A(k, j), A(i, j))
        }
    }
}
```

Figure 4.24: Classic Cholesky factorization using loops and basic kernels.

All the relationships between tasks are described wiht these algebraic expressions which also connect input-output data dependencies between tasks. A task can independently query the parameter-range expressions for complete information about its data communication and its relationships to other tasks. The size of the PTG is related to the number of task classes, and not to the size of the problem being solved, so this compact representation can easily be evaluated at each computing node, to provide details of both local and remote operations.

From the point of view of the programmer, once a PTG has been defined, PaRSEC schedules the tasks onto the hardware and handles the data communication transparently. The binding of tasks to distributed memory nodes is determined at the task insertion time. By default this binding is determined by the data layout, but the programmer can create other task bindings at will. Within each node, the scheduling and execution of the tasks can be dynamically managed at execution time, allowing PaRSEC to do node level load balancing. The communication is managed by PaRSEC, and can be overlapped with computation. In the current state, job stealing, to reduce load imbalance among nodes, is not implemented yet.

Sequential Task Flow

The PaRSEC team is working on adding the Sequential Task Flow (STF) interface, to provide a simpler alternative to writing applications using the PTG. Tasks are added to the DAG

```
POTRF(k) [high_priority = on]
// Execution space
k = 0 .. descA.mt-1
// Parallel partitioning
:dataA(k, k)
// Parameters
RW T <- (k == 0) ?
        dataA(k, k) : T SYRK(k-1, k)
     -> T TRSM(k+1..descA.mt-1, k)
     -> dataA(k, k)
BODY
{
    int tempkm = k == descA.mt-1 ?
        descA.m - k*descA.mb : descA.mb;
    int iinfo = 0;
    int ldak = BLKLDD( descA, k );

    CORE_dpotrf(
        uplo, tempkm, T, ldak,
        &iinfo );
    if ( iinfo != 0 && *INFO == 0 )
        *INFO = k*descA.mb+iinfo;
}
```

(a) Factor a diagonal block.

```
SYRK(k, m) [high_priority = on]
// Execution space
k = 0   .. descA.mt-2
m = k+1 .. descA.mt-1
// Parallel partitioning
: dataA(m, m)
//Parameters
READ  A <- C TRSM(m, k)
RW    T <- (k == 0) ?
           dataA(m, m) : T SYRK(k-1, m)
        -> (m == k+1) ?
           T POTRF(m)  : T SYRK(k+1, m)
BODY
{
    int tempmm = m == descA.mt-1 ?
        descA.m - m*descA.mb : descA.mb;
    int ldam = BLKLDD( descA, m );

    CORE_dsyrk(
        PlasmaLower, PlasmaNoTrans,
        tempmm, descA.mb,
        (double)-1.0, A /*A(m, k)*/, ldam,
        (double) 1.0, T /*A(m, m)*/, ldam);
}
END
```

(b) Update a diagonal block.

```
TRSM(m, k) [high_priority = on]
// Execution space
m = 1 .. descA.mt-1
k = 0 .. m-1
// Parallel partitioning
: dataA(m, k)
// Parameters
READ  T <- T POTRF(k)
RW    C <- (k == 0) ?
           dataA(m, k) : C GEMM(m, k, k-1)
        -> A SYRK(k, m)
        -> A GEMM(m, k+1..m-1, k)
        -> B GEMM(m+1..descA.mt-1, m, k)
        -> dataA(m, k)
BODY
{
    int tempmm = m == descA.mt-1 ?
        descA.m - m * descA.mb : descA.mb;
    int ldak = BLKLDD( descA, k );
    int ldam = BLKLDD( descA, m );

    CORE_dtrsm(
        PlasmaRight, PlasmaLower,
        PlasmaTrans, PlasmaNonUnit,
        tempmm, descA.nb,
        (double)1.0, T /*A(k, k)*/, ldak,
                     C /*A(m, k)*/, ldam);
}
END
```

(c) Factor an off-diagonal block.

```
GEMM(m, n, k)
// Execution space
k = 0   .. descA.mt-3
m = k+2 .. descA.mt-1
n = k+1 .. m-1
// Parallel partitioning
: dataA(m, n)
// Parameters
READ  A <- C TRSM(m, k)
READ  B <- C TRSM(n, k)
RW    C <- (k == 0) ?
           dataA(m, n)  : C GEMM(m, n, k-1)
        -> (n == k+1) ?
           C TRSM(m, n) : C GEMM(m, n, k+1)
BODY
{
    int tempmm = m == descA.mt-1 ?
        descA.m - m * descA.mb : descA.mb;
    int ldam = BLKLDD( descA, m );
    int ldan = BLKLDD( descA, n );

    CORE_dgemm(
        PlasmaNoTrans, PlasmaTrans,
        tempmm, descA.mb, descA.mb,
        (double)-1.0, A /*A(m, k)*/, ldam,
                      B /*A(n, k)*/, ldan,
        (double) 1.0, C /*A(m, n)*/, ldam);
}
END
```

(d) Update an off-diagonal block.

Figure 4.25: PaRSEC Cholesky factorization subroutine based on JDF. The relationships between the four kernel routines and the dataflow between them is expressed in the “Parameters” clause.

sequentially, with the parameters tagged as read and/or write. The runtime can then use the sequential task insertion order, and the data tags, to determine all the dependencies and execute the workload correctly. This technique is often referred to as *task-superscalar*. Figure 4.26 shows the PaRSEC implementation of the Cholesky factorization using the insert-task interface.

In distributed systems there are limitations to the scalability of the STF approach. The entire DAG must be discovered sequentially, by each node, i.e., each node must track all the dependencies, even the remote ones.

```
for( int k = 0; k < n; k++) {
    insert_task(handle, &kernel_potrf, priority, "POTRF",
        PASSED_BY_REF, TILE_OF(A, k, k, 0), INOUT | REGION_FULL | AFFINITY,
        sizeof(int), &ldak, VALUE,
        0);
    for (int j = k+1; j < n; j++)
        insert_task(handle, &kernel_trsm, priority, "TRSM",
            PASSED_BY_REF, TILE_OF(A, k, k, 0), INPUT | REGION_FULL,
            PASSED_BY_REF, TILE_OF(A, k, j, 0), INOUT | REGION_FULL | AFFINITY,
            0);

    for( int i = 1; i <= n; i++ ) {
        insert_task(handle, &kernel_syrk, priority, "SYRK",
            PASSED_BY_REF, TILE_OF(A, k, i, 0), INPUT | REGION_FULL,
            PASSED_BY_REF, TILE_OF(A, i, i, 0), INOUT | REGION_FULL | AFFINITY,
            0);
        for (int j = i+1; j <= n; j++) {
            insert_task(handle, &kernel_gemm, priority, "GEMM",
                PASSED_BY_REF, TILE_OF(A, k, i, 0), INPUT | REGION_FULL,
                PASSED_BY_REF, TILE_OF(A, k, j, 0), INPUT | REGION_FULL,
                PASSED_BY_REF, TILE_OF(A, i, j, 0), INOUT | REGION_FULL | AFFINITY,
                0);
        }
        dtd_data_flush(handle, TILE_OF(A, k, i, 0));
    }
}
```

Figure 4.26: PaRSEC Cholesky factorization using the Sequential Task Flow (insert-task) interface.

Accelerator Support

PaRSEC supports the use of accelerators, such as GPUs, and handles the communication to these devices transparently. At present, the use of these devices is specified in the PTG at compile time, rather than being handled automatically at rumtime. In order to utilize the computing power of GPUs efficiently, current efforts target support for "hiearchical DAGs", where smaller DAGs can be nested within larger ones.

What does PaRSEC do best?

- The compact DAG representation in PaRSEC (PTG) avoids many bottlenecks in the generation and tracking of task dependencies. At the same time, the task-superscalar scheduling, while much more convenient to use, is limited in scalability by the process of sequential DAG unrolling.

- PaRSEC minimizes the serial overheads and has good scalability to large numbers of distributed memory nodes [10].
- PaRSEC is able to can use accelerators semi-transparently using kernels that are built for the accelerator. However, the current scheduling mechanism does not abstract execution to allow PaRSEC to dynamically choose accelerators versus CPUs, depending on the executing workload. This decision is statically made by the programmer.

Where does PaRSEC not do well?

- Developing a compact DAG representation for an application is a challenging task, which quickly gets more difficult with the complexity of the dependencies in the algorithm.
- PaRSEC does not easily support data-dependent DAGs. For example, computations with “if” conditions depending on the data are not be supported. Workarounds are usually possible.
- PaRSEC does not mix wiht synchronization techniques outside of its paradigm. For instance, the LU factorization with partial pivoting requires an implementation of of the panel factorization operation using a “gang” of tasks, synchronized using barriers. Currently, PaRSEC does not facilitate such implementations, which has been a major roadblock for building an afficient implementation of the LU factorization.
- The tasks are not dynamically scheduled over all the distributed memory nodes at execution time. There are some positive aspects to this, since the long term load balancing may be better with the static binding, and data movement can be initiate early.

PaRSEC Performance at Large Scale

PaRSEC has shown excellent performance and scalability. PaRSEC implementation of the QR factorization of a $M = N = 41,472$ matrix achieved performance many times higher then that of the Cray LibSci library, when ran using $23,868$ cores of the Kraken supercomputer at ORNL (Cray XT5) [10]. PaRSEC has also been shown to perform well on distributed-memory hybrid, accelerator-based, machines [85].

Pros of using Parsec for SLATE:

- Offers good scaling properties with PTGs.
- Takes care of scheduling work to accelerators.
- Handles asynchronous tasks scheduling and management.
- Enables deep lookaheads because it “sees” the entire DAG at once.

Cons of using Parsec for SLATE:

- Complex algorithms are not easily expressed in the PTG notation.
- Does not mix well with other scheduling paradigms.
- Does not facilitate data-dependent scheduling.

4.2.2 Legion

Legion is a data-centric parallel programming system that deals primarily with inter-node parallelism. Legion is a C++ framework (template, runtime, library) that provides annotations and scheduling of data dependencies, and allows customized mapping from tasks/data to hardware execution unit and memory. "Logical Region" is the core abstraction in Legion to describe the structure of program data; "Mapping Interface" is the core mechanism to map program data to physical memory and tasks to execution units.

Logical Region

A Legion program first decomposes the computation objects into logical regions. Regions can be further partitioned into sub-regions. Regions can even overlap with each other. Functions that specify which regions to touch, and the priviledges/coherence associated with the regions, are called *tasks*. The Legion runtime features a software out-of-order scheduler performs parallel execution of tasks that honors data dependencies. Figure 4.27 shows an example of a circuit simulation reported by Bauer et al. [13]. There are four points of interests in this example:

1. (line 5) In the task specification, the regions (data) that the task is going to touch are marked with privilege and coherence. `RWE` means Read-Write-Exclusive, i.e., the task is going to read and write the regions; the coherence is exclusive meaning the execution order is to be maintained.
2. (line 7 to 9) This is where data is partitioned into logical regions. The partition can be marked as `disjoint` or `aliased`, depending on whether the partitions can overlap. Tasks working on disjoint data can be run in parallel, while tasks working on shared (aliased) data might require communication and synchronization.
3. (line 14) This is how the tasks are issued, in this case following three phases in each time step. Note that the tasks are not necessarily run according to the issuing order; the Legion scheduler will schedule the tasks concurrently and out of order, as long as the data dependency allows. Also, note that there is no explicit synchronization or communication involved.
4. (line 21 to 24) This is a task working on a region that consists of a private region, a shared region, and also an aliased ghost region. This task will update the shared region and the aliased region, which poses as potential conflict for other tasks. Thus, the task is marked as RdA (Reduction-Atomic) to enable concurrent update to the aliased regions.

```
struct Node {...};
struct Wire {...};
struct Circuit { region r_all_nodes; region r_all_wires; };

void simulate_circuit(Cicuit c, float dt) :  RWE(c.r_all_nodes, c.r_all_wires)
{
        partition<disjoint> p_wires = c.r_all_wires.partition(wire_map);
        ...
        partition<aliased> p_ghost_nodes = p_nodes_pvs[1].partition(node_neighbor_map);


        ...
        for (t=0; t<TIME_STEPS; t++) {
                spawn(i=0; i<MAX_PIECES; i++) distribute_charge(pieces[i],dt);
                ...
        }

}

void distribute_charge(CircuitPiece piece, float dt):
        ROE(piece.rw_pvt), RdA(piece.rn_pvt, piece.rn_shr, piece.rn_ghost) {
        foreach (w: piece.rw_pvt)
                w->in_node->new_charge += -dt * w->current;
                w->out_node->new_charge += dt * w->current;
        }
}
```

Figure 4.27: A Circuit Simulator in Legion pseudo code (excerpt adapted from the article by Bauer et al. [13]).

The *mapping interface* gives the program control over where tasks run and where region instances are placed (but when to run the tasks is determined by the SOOP scheduler). The interface is invoked at runtime, which enables adaptive mapping based on input data. There are three most important mapping interfaces: `select_initial_processor`, `permit_task_steal`, `map_task_region`. There is a default mapper that has default policies for these interfaces. However, the essential flexibility comes from the ability to customize the mapper, i.e., overriding certain aspects of the default mapper. The mapping can be customized to be completely static, fully dynamic, or something in between.

Tasks

In Legion, the *task* is the construct to describe computation. Tasks are asynchronous and annotated with the regions that they access. The Legion runtime is responsible to schedule the execution of tasks, and to maintain the sequential semantics of the Legion program, under the constraints of data dependencies and other synchronization directives. In addition to the task specification and launch examples shown in Figure 4.27, tasks have more features and nuances, as shown by the example in Figure 4.28 (adapted from Section 2.4 in the article by Bauer et al. [14]).

The following new concepts emerge from the Conjugate Gradient example in Figure 4.28:

Sub-tasks: A Legion program execution is carried out by a tree of tasks. During execution, tasks can launch sub-tasks, with the *containment property*, which dictates that sub-

```
struct SparseMatrix {
  region lr;
  partition<disjoint> part;
  int n_rows, elmts_per_row;
}
struct Vector {
  region lr;
  partition<disjoint> part;
  int n_elmts;
}

void CG(SparseMatrix a, Vector x): RWE(a.lr, x.lr) {
  tunable int num_pieces;
  a.part = a.lr.partition(num_pieces);
  x.part = x.lr.partition(num_pieces);
  Vector r_old(x.n elmts), p(x.n elmts), A_p(x.n elmts);

  spawn<num pieces> spmv(a.part, x.part, A_p.part);
  spawn<num pieces> subtract(b.part, A_p.part, r_old.part);
  double L2Norm0 = spawn<num_pieces> L2norm(r_old.part);
  copy(r_old, p);

  predicate loop_pred = true;
  future r2_old, pAp, alpha, r2_new, beta;
  for (...) {
    spawn<num pieces> @loop_pred spmv(A.part, p.part, A_p.part);
    r2_old = spawn<num_pieces><+> @loop_pred dot(r_old.part, r_old.part, r2_old);
    pAp = spawn<num_pieces><+> @loop_pred dot(p.part, A_p.part, alpha);
    alpha = spawn @loop_pred divide<r2_old,pAp);
    spawn<num_pieces> @loop_pred daxpy(x.part, p.part, alpha);
    spawn<num_pieces> @loop_pred daxpy(r_old.part, A_p.part, -alpha);
    r2_new = spawn<num_pieces><+> @loop_pred dot(r_old.part, r_old.part,r2_new);
    beta = spawn @loop_pred daxpy(r_old.part, p.part, beta);
    future norm = spawn<num_pieces><+> @loop_pred dot(r_old.part,r_old.part,L2norm);
    loop_pred = spawn @loop_pred test_convergence(norm, L2norm) : false;
  }
}
```

Figure 4.28: A Conjugate Gradient linear solver implementation in Legion. Pseudo-code adapted from [14].

tasks can only access a subset of the regions accessible from the parent task, with the privileges equal or inferior to the privileges of the parent task. This requirement eases the scheduling of the task trees.

Index Space Tasks: Legion provides the mechanism to launch many sub-tasks simultaneously through *Index Space Tasks Launch*. This can reduce runtime overhead of launching many tasks, and also provide the opportunity to express properties of group tasks, such as the ability to synchronize within a group.

Futures: As seen in the CG example in Figure 4.28 Legion supports a construct called a *future*. This is a similar construct to futures in many other languages, such as the C++11, and indicates a value that promises to be available sometime in the future. A task can wait on the availability of a future value, i.e., using it which will block the task. Better yet, it can pass the future along, into sub-tasks, to avoid blocking (line 34, 35).

Predicated Execution: Legion allows sub-task launches to be predicated on a boolean value that is not resolved. In the CG example in Figure 4.28 the main iteration (lines 25-36) will stop based on the result of `test_convergence()`. However the next iteration will not wait until the result of the `test_convergence()` is available. The new tasks will be spawned and allows the analysis of the task dependencies to not block on the availability of the test result. Whether the execution of the tasks can be predicated is a separate issue.

Task Variants and Qualifiers: Legion allows multiple variants of the same task to support optimization in differing execution environments. Tasks can also be tagged with the qualifiers: *leaf*, *inner*, and *idempotent*. Leaf tasks do not generate sub-tasks. Inner tasks on the contrary, do nothing but generate sub-tasks. These two qualifiers aid in the analysis of the dependencies and scheduling. An idempotent task has no side effects, except for the declared regions. This qualifier helps with resilience.

What does Legion do and not do?

The central role of Legion is to schedule tasks in a way that preserves “locality” and “independence”. Legion deals with *when* to run the tasks, while leaving the question of *where* to run the tasks, and *where* to place the data, to the user.

Legion does not automatically generate tasks. Legion does not automatically map tasks/data to hardware. Legion does not automatically decompose the program. Legion does not put the first priority on productivity.

Performance of Legion at large scale

In the article by Bauer et al. [14] a production-quality combustion simulation S3D was ported to Legion and demonstrated to be 3× faster than state-of-the-art S3D written in MPI/OpenACC, when run using 8,192 nodes.

Implications in using Legion for SLATE:

- Legion depends on GASNet for inter-node communication. There is no explicit communication involved, thus the user has no direct control over communication.
- If SLATE uses Legion, will applications have to use Legion as well for parallelizing their other components?
- One of the desirable feature of Legion is the ability to control where to run tasks and where to place data, which is essential in obtaining high performance.
- The dynamic out-of-order task scheduler seems appealing for numerical linear algebra at large scale on heterogeneous nodes.
- Legion seems to struggle with performance when running load balanced applications on homogeneous machines (see page 95 in the Sandia report by Bennett et al. [15]).

4.2.3 DARMA

DARMA was a response to the 2015 study by Bennett et al. Bennett et al. [15] in assessing the leading Asynchronous Many-Task (AMT) runtimes. The report studies extensively the three major AMT systems, Charm++, Legion, and Uintah, in the context of ASC applications. Charm++ implements a low-level actor model and replaces MPI messages with remote procedure calls. Legion is a data centric model with declarative program expression (see Section 4.2.2). Uintah is a scientific domain-specific system for PDE on structured grids. Uintah is too domain-specific, and will not be discussed further. It is instructive to read the comparative study on the performance, expressive style, programmability, and scalability of the three AMT runtime systems in the context of the MiniAero application. The report provides three primary conclusions:

- AMT systems are promising for large scale heterogeneous computing.
- APIs of the AMT systems vary.
- There is a need for identifying best practices and standards.

Thus DARMA was created to provide a unified AMT API that (potentially) maps to multiple backends (Legion, Charm++, etc.) for providing a single unified API, and for studying AMT best practices, specifically for Sandia's requirements. As of the writing of this document it maps to Charm++.

DARMA can be considered as a two layer system as shown in Figure 4.29: the frontend and backend. The frontend interfaces with the application by providing abstractions to express data-task dependencies. The backend consists of glue code and runtime system to smoothly map to existing AMT systems (Charm++, Legion) for scheduling and executing the tasks.

The core abstractions of the frontend is the *data* abstraction and the *task* abstraction. Data are wrapped with the `darma::AccessHandle<T>` or `darma::AccessHandleCollection<T>` constructs. Tasks are created using `darma::create_work` or `darma::create_concurrent_work`,

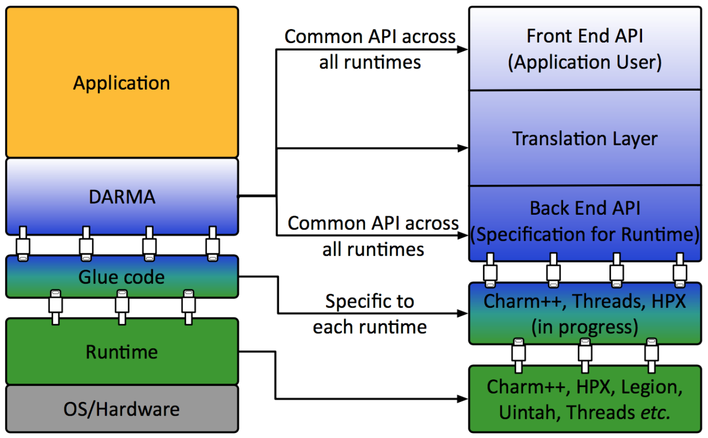

Application
DARMA
Glue code
Runtime
OS/Hardware
Common API across all runtimes
Common API across all runtimes
Specific to each runtime
Front End API (Application User)
Translation Layer
Back End API (Specification for Runtime)
Charm++, Threads, HPX (in progress)
Charm++, HPX, Legion, Uintah, Threads *etc.*

Figure 4.29: The structure of DARMA (source: 2017 Sandia slides by Markosyan et al.).

with the code for the task encoded in a C++ lambda function or a functor. These constructs enforce sequential semantics, which is commonly assumed by the programmer if no parallel processing occurs. The system extracts concurrency by constructing a data-task dependency DAG and scheduling the execution of tasks in parallel, much like an out-of-order processor does for instructions. An example illustrating the use of these constructs and the execution order is shown in Figure 4.30. Note that the 2nd and 3rd tasks have the annotation that they only read `my_data`; the other two tasks default to updating `my_data`. It turns out that task 2 and 3 can execute in parallel, after the execution of task 1, and before the execution of task 4.

```
AccessHandle<int> my_data;
darma::create_work([=]{
  my_data.set_value(29);
});
darma::create_work(
  reads(my_data), [=]{
    cout << my_data.get_value();
  }
);
darma::create_work(
  reads(my_data), [=]{
    cout << my_data.get_value();
  }
);
darma::create_work([=]{
  my_data.set_value(31);
});
```

Figure 4.30: An example of DARMA code (source: 2017 Sandia slides by Bennett et al.).

An important extension to the scalar data abstraction `AccessHandle` is the collection of data abstractions `AccessHandleCollection`. An example of this collection object is shown in Figure 4.31. In this case, the `mycol` variable will be a collection of `vector<double>`, with an index range `Range1D(10)` meaning the elements in the collection can be accessed via indices 0-9. An associated extension deals with creating a collection of tasks with access to a collection of data. Each task is similarly created with an index range, and the programmer decides the elements in the data collection to use in the task, based on its task index.

```
    AccessHandleCollection<vector<double>, Range1D> mycol =
    darma::initial_access_collection(index_range=Range1D(10));
```

Figure 4.31: Example of the AccessHandleCollection construct.

DARMA supports another interesting communication pattern, other than passing data access handles to tasks, a pub/sub semantic called publish/fetch in DARMA terminology. When creating access handles for the data a string "key" can be attached. The access handle can be "published" using its `publish()` method with a version string. Other tasks can create access an handle to the same piece of data using its "key" name and version, via the `read_access()` method. Thus the publish/fetch semantics seem to create a distributed key-value space.

4.2.4 Kokkos

Kokkos [38] is a performance portability layer for intra-node parallel programming. In this regards, it competes with OpenMP and OpenACC. However, Kokkos is not a language extension; it is a C++ templated library and a runtime. The unique characteristic of Kokkos is the combination of both parallel execution abstraction and architecture optimized memory layout multi-dimensional array abstraction. Although both Kokkos and OpenMP/OpenACC allow for parallelizing (serial) loops, Kokkos relies on *parallel patterns* instead of loops. Kokkos maps the parallel patterns into a serial loop for single thread, multithreaded loops for multiple threads, or GPU threads for a GPU. Here is an example of the parallel-for pattern:

```
// Serial
for (int i=0;i<n;i++) {
  // loop body
}

// OpenMP
#pragma omp parallel for
for (int i=0;i<n;i++) {
  // loop body
}

// Kokkos
parallel_for(n, [=] (const int i) {
  // loop body
});
```

Besides the parallel_for pattern, Kokkos also has the reduction pattern parallel_reduce, and also parallel_scan. It also seems to include DAG task pattern, but we have not seen it discussed anywhere, other than Kokkos SC'16 tutorial slides.

The central claim of Kokkos is that it can achieve portable performance over multi-core, many-core, and GPU systems through a multi-dimensional array abstraction called a *View*. Figure 4.32 shows a declaration of a 4-dimensional view. Depending on the execution target, the array abstraction will have different data layout in memory. The layout is determined at compile time. Like in RAJA, the serial a loop in a program needs to be expressed using one of the *patterns* (for, reduce, task-graph), and the loop body needs to be expresses using lambdas or functors in C++. The work then is mapped to threads according to the execution target (mapping indexes in contiguous chunks on CPUs, and strided on GPUs).

```
// The following declares a N*M*8*3 array with both
// runtime and compile time determined dimensions.
View<double**[8][3],Device> a("A",N,M);

// 8x8 tiled layout used in PLASMA
View<double**, LayoutTileLeft<8,8>, Device> b("B",N, M);

a(i,j,k,l) = value;
```

Figure 4.32: Declaration of a multidimensional array (View) in Kokkos.

Kokkos introduces the concept of an *execution space* and a *memory space*. The execution space indicates where the code is executed (CPU or GPU), while the memory space indicates where the array (View) is stored (host memory or device memory).

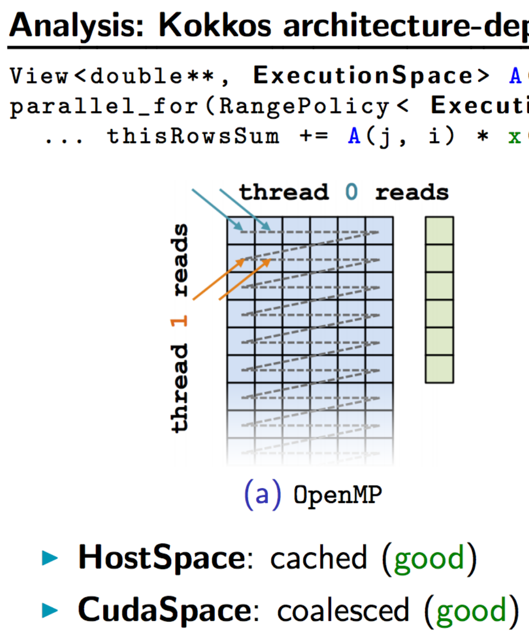
Execution Space: Heterogeneous nodes have one or more execution spaces (host, device). The programmer controls where the code is run by a template parameter for the execution policy. The execution place of a certain piece of code is thus determined at compile time.

Memory Space: Heterogeneous nodes have one or more memory spaces (HostSpace, CudaSpace, CudaUVMSpace). The code that runs in the HostSpace cannot directly access views from CudaSpace for example; there are two solutions: 1) declare the views in CudaUVMSpace instead of CudaSpace; 2) create a “mirror” of the desired view in a different memory space. The former will likely suffer from bad performance, as the runtime may have to handle suboptimal data movement; the latter takes more space and requires manual data copies.

Efficient memory access is then achieved by Kokkos by mapping parallel work and multidimensional array layout optimally to the architecture (see Figure 4.33). Every View has a Layout specified at compile time through a template parameter. LayoutRight (C array style) and LayoutLeft (Fortran array style) are the most common, although the layout can be arbitrarily customized (tile layout for example, as used in PLASMA [24]). For performance, the memory access pattern on CPUs should allow for good caching, while on GPUs it should allow for good coalescing. Kokkos allows for mapping multidimensional arrays to assure efficient memory access for each type of target hardware. In contrast, OpenMP/OpenACC/OpenCL has no notion of data layout customization; multiple versions of the code must be maintained for execution on CPUs and GPUs.

Memory Traits

Beyond *Layout* and *Space*, Views can have memory traits, such as Atomic, Read-only, and Random. These traits indicate the access pattern to the view, thus allowing for hardware-specific optimizations to be performed. For example, views with the Atomic trait can be instantiated using atomic instructions on a supported architecture. Views with read-only trait can be put into texture memory on GPUs.



Analysis: Kokkos architecture-dependent

```
View<double**, ExecutionSpace> A(N, M);
parallel_for(RangePolicy< ExecutionSpace>(0, N),
  ... thisRowsSum += A(j, i) * x(i);
```

- **HostSpace**: cached (good)
- **CudaSpace**: coalesced (good)

Figure 4.33: Different mapping of the data layout for a CPU and a GPU in Kokkos. (source: https://github.com/kokkos/kokkos-tutorials/blob/master/SC2016/KokkosTutorial_SC2016.pdf)

```
template<typename T> T atomic_exchange(T *dest, T val);
template<typename T> bool atomic_compare_exchange_strong(T *dest, T comp, Tval);
```

Figure 4.34: Atomic exchange for arbitary data types in Kokkos.

Scalable Thread Safety with Atomics

Perhaps the most popular synchronization primitives used in low-count CPU threading is *locking*. The performance however is not satisfactory for scaling to many-core systems. Kokkos provides atomic operations (`Kokkos::atomic_add()`) as a scalable thread safety solution. The canonical example, where such synchronization is needed, is a multi-threaded histogram update. A typical scenario is one, where each thread iterates through a portion of the data, and updates the corresponding bins. Multiple threads might try to update the same bin, thus creating a race condition. The Kokkos atomic operation provides thread safety with very low overhead, at low contention, by using backend atomics (OpenMP, CUDA, Phi) whenever possible. The atomics only exist for certain data types though. Another synchronization primitive *atomic exchange* exists for any data type (figure 4.34).

Hierarchical Parallelism

Node architectures of modern HPC systems are characterized by hierarchical parallelism. For example, in a multi-socket CPU systems, there are 4 levels of parallelism:

1. multiple CPU sockets in each node, sharing the same memory;
2. multiple cores in each socket, typically sharing the last level cache (LLC);
3. mulitple hyperthreads in a core sharing the L1/L2 caches and some functional units;
4. multiple SIMD lanes sharing instructions.

The situation is similar for NVIDIA GPUs, where multiple levels of parallelism exist:

1. multiple GPUs in a node, sharing the unified address space;
2. multiple SMs in a GPU, sharing the same *device memory*;
3. multiple waprs in each SM, sharing registers and caches;
4. multiple threads in each warp, sharing the same instruction stream.

As the architectures features hierarchical parallelism, the parallel programming system should also provide hierarchical abstractions to efficiently exploit the hardware. The core concept in Kokkos is called *thread teams*. A thread team is a collection of threads which can synchronize, and share a scratchpad memory, e.g., shared memory in CUDA. In Kokkos there are three levels of parallelism: team level, thread level, and vector level.

The most basic hierarchical parallelism concept is the *thread team*, which is a team of threads that can synchronize and access a shared scratchpad memory. Kokkos' thread thus is identified by two indexes: the *league* index, which identifies the team, and the thread index, which identifies the thread within the team. This arrangement is analogous to the 1-D grid of 1-D blocks in the CUDA nomenclature. Similarly to CUDA, threads from different teams do not synchronize or share scratchpad memory.

Consider the following example. Suppose that we want to calculate the inner product $y^T Ax$. We could employ two level nested parallelism here: each team is assigned a row of A, and each thread in assigned a column in that row (Figure 4.35). We can see that there is a nested `parallel_reduce()` inside the main `parallel_reduce()`, one using `team_policy` and the other using `TeamThreadRange`. Again, within a team, the mapping between indexes and threads should be abstracted, which allows architecture dependent policies – contiguous mapping on CPUs and strided mapping on GPUs. Note that the inner parallel policy is always `TeamThreadRange` and cannot be further nested.

Another aspect of hierarchical parallelism is the *scratchpad memory* that is private to a team of threads.

```
parallel_reduce(
  team_policy(N, Kokkos::AUTO),
  KOKKOS_LAMBDA(member_type & teamMember,double &update) {
    int row = teamMember.league_rank();
    double thisRowsSum = 0;
    parallel_reduce(TeamThreadRange(teamMember, M),
      [=] (int col, double &innerUpdate) {
        innerUpdate += A(row, col) * x(col);
      }, thisRowsSum);
    if (teamMember.team_rank() == 0) {
      update += y(row) * thisRowsSum;
    }
  }, result);
```

Figure 4.35: Two-level parallel inner product in Kokkos

DAG Tasking

It seems that the DAG tasking is a 2017 June milestone for the Kokkos project. At the time of writing this document, support for DAG tasking is unclear.

Implications of using Kokkos for SLATE

Kokkos provides polymorphic data layout in multi-dimensional array to support portable performance across CPUs and GPUs. However, optimal performance may not only depend on data layout but also on algorithms. The simple execution pattern abstractions in Kokkos (parallel for/reduce/scan) might only go so far in expressing more involved algorithms. As such, multiple versions of code or code paths still are needed to achieve portable performance.

4.2.5 RAJA

RAJA is a portability layer that leverages the fine-grained parallelism at the node level (similar to OpenMP, OpenACC, CUDA, etc.), with a cross-platform support. RAJA was initially developed for the large ASC hydrodynamics codes at LLNL (LULESH, Ares, Kull, and ALE3D), and is, therefore, primarily tailored to the problem structures and parallelization challenges in these codes. The fundamental conceptual abstraction in RAJA is an inner loop, where the overwhelming majority of computational work in most physics codes occurs. These loops can then be executed in a parallel fashion, using the available resources. The main features of RAJA, which are discussed in detail later on, are as follows:

1. RAJA uses an abstract *execution policy* for loop execution. An execution policy is a template parameter that encapsulates the details of the loop execution, e.g. sequential execution, parallel execution, enable SIMD, etc. Since the description of different execution policies exists in the headers, RAJA codes can easily switch between execution policies without retouching the loop body.

2. RAJA uses *IndexSets* to partition the iteration space and handle data placement. An *IndexSet* is an object that encapsulates a complete iteration space, which is partitioned

into a collection of segments, of the same or different types. RAJA IndexSets are similar to the iterators available in the LUA language.

3. RAJA uses C++ lambda functions, which enable capturing the loop body without modification.
4. To hide non-portable compiler directives and data attributes, RAJA uses *data type encapsulation*, e.g., “Real_type” and “Real_ptr” instead of “double” and “double*.”
5. RAJA requires the C++11 standard.

Porting Existing Software to RAJA

A typical RAJA integration approach involves the following steps:

1. Transform the loops to be RAJA-enabled. This is a straightforward process in most cases. Such initial transformation makes the code portable by enabling it to execute on both the CPU and the GPU by choosing various parallel programming model back-ends at compile-time.
2. Choose the execution policy. The choice(s) can be refined based on an analysis of loops. Careful categorization of loop patterns and workloads is key to selecting the best choices for mapping loop execution to available hardware resources for high performance. Important considerations include:
   (a) The arithmetic intensity of the operation executed by the loop.
   (b) The existence of control flow and branching operations.
   (c) The available parallelism across different iterations.
3. If required, a deeper analysis of the algorithm can lead to utilizing more advanced features of RAJA that are platform-specific. Such advanced mechanisms can be implemented in RAJA transparently, and then propagated to all codes that have a similar pattern and target the same platform.

Decoupling Loop Body from Loop Traversal

RAJA relies on separating the body of a loop from the mechanism that executes it (its traversal). This allows the same traversal method to be an abstraction that is applicable to many different loop bodies. It also allows different traversals to be applied to the same loop body for different execution scenarios. In RAJA, the decoupling is achieved by recasting a loop into the generally-accepted ”parallel for“ idiom. As an example, Figure 4.36 shows an example for a simple C++ loop and its RAJA equivalent.

There are several key differences to note in the RAJA loop shown in Figure 4.36:

1. The `for` loop construct is replaced by a call to a traversal template method (`RAJA::forall`), where the template parameter is the loop execution policy.

```
double* x ; double* y ;
double a , tsum = 0 . 0 , tmin = MYMAX;
/* some code */
for ( int i = begin ; i < end ; ++i ) {
    y[i] += a * x [i] ;
    tsum += y [ i ] ;
    if( y[i] < tmin ) tmin = y[i];
}
```

```
double* x; double* y;
RAJA::SumReduction<reduce_policy, double> tsum(0.0);
RAJA::MinReduction<reduce_policy, double> tmin(MYMAX);
/* some code */
RAJA::forall<execpolicy>(begin, end, [=](int i){
    y[i] += a * x[i];
    tsum += y[i];
    tmin.min( y[i] );
}
```

Figure 4.36: A RAJA equivalent to a simple loop in C++.

2. The loop body is passed to the traversal template as a C++ lambda function.
3. The reduction variables are converted to RAJA objects with templated reduction policy and reduction data type.

It is important to note that the original `for` loop explicitly expresses all the execution details in the source code, such as iteration order and data accesses. Changing any aspect of execution requires changes to this source code. Decoupling the loop body from traversal as in the RAJA version, iteration orders, data layout and access patterns, parallel execution strategies, etc. can be altered without changing the way the loop is written. Apart from the slight difference in syntax for the `min` reduction, the loop body is the same as the C-style version. The C++11 lambda function capability enables the key RAJA design goal, which is to achieve portability with minimal disruption to the application source code.

RAJA Encapsulation Model

Figure 4.37 describes four main encapsulation features in RAJA, each with a different color, that can be used to manage architecture-specific concerns.

```
RAJA::Real_ptr x, RAJA::Real_ptr y;
RAJA::Real_type a;
RAJA::SumReduction<..., Real_type> tsum(0);
RAJA::MinReduction<..., Real_type> tmin(MYMAX);

RAJA::forall<exec_policy>( IndexSet, [=]( Index_type i) {
    y[i] += a * x[i];
    tsum += y[i];
    tmin.min( y[i] );
} );
```

Figure 4.37: Different encapsulations in RAJA.

Traversals and execution policies [blue]: A traversal method, specialized with an execution policy template parameter, defines how the loop will be executed. For example, a traversal may run the loop sequentially, as multithreaded parallel loop using OpenMP, or may launch the loop iterations as a CUDA kernel to run on a GPU.

IndexSets [purple]: Figure 4.36 shows that the `begin` and the `end` loop bounds are passed as arguments to the traversal method. While RAJA can process explicitly bounded loop iterations in various execution schemes that are transparent to the source code, the RAJA *IndexSet* abstraction in Figure 4.37 enables much more flexible and powerful ways to control loop iterations. IndexSets allow loop iteration order to be changed in ways which can, for example, enable parallel execution of a non-data-parallel loop without rewriting it. Typically, an IndexSet is used to partition an iteration space into *segments*; i.e., "chunks" of iterations. Then, different subsets of iterations may be launched in parallel or run on different hardware resources. IndexSets also provide the ability to manage dependencies among segments to resolve thread safety issues, such as data races. In addition, IndexSet segments enable coordination of iteration and data placement; specifically, chunks of data and iterations can be mapped to individual cores on a multi-core architecture. While IndexSets provide the flexibility to be defined at runtime, compilers can optimize execution of kernels for different segment type implementations at compile-time.

Data type encapsulation [red]: RAJA provides data and pointer types, that can be used to hide non-portable compiler directives and data attributes, such as alignment, restrict, etc. These compiler-specific data decorations often enhance the compiler's ability to optimize the code. For any parallel reduction operation, RAJA requires a reduction class template to be used. Template specialization of a reduction enables a portable reduction operation while hiding the programming of model-specific reduction constructs from application code.

C++ lambda functions [brown]: The standard C++11 lambda feature captures all variables used in the loop body which allows the loop construct to be transformed, with minimal modification, to the original code.

The RAJA encapsulation features described here can be used individually or combined together, depending on the portability and performance needs of the application. They may also be combined with application-specific implementations. This allows a multi-tiered approach to performance tuning for a particular architecture. Most loops in a typical HPC application can be parameterized using basic RAJA encapsulation features. Other kernels may require a combination of RAJA entities and customized implementations suited to a particular algorithm.

Basic Traversal Methods and Execution Policies

We consider the code example of Figure 4.36. Since the execution policy is passed as a template parameter, the same loop can be executed in different ways. We assume that the policy template parameters are defined as `typedefs` in a header file.

A CPU serial execution can be realized using one of RAJA's built-in execution policies. This requires the following definition:

```
typedef RAJA::sequential    exec_policy;
typedef RAJA::seq_reduce    reduce_policy;
```

Such definition leads to a traversal template that looks like:

```
template<typename LB>
void forall(sequential, Index_type begin, Index_type end, LB body){
    #pragma novector
    for (int i = begin; i < end; ++i) body(i);
}
```

Note that the `novector` pragma option prevents the compiler from generating SIMD vectorization optimizations for this case. Changing `exec_policy` to `RAJA::simd` allows the compiler to generate SIMD optimizations if it decides to do so.

The following definition leads to a parallel CPU execution using OpenMP:

```
typedef RAJA::omp_parallel_for exec_policy;
typedef RAJA::omp_reduce       reduce_policy;
```

It tells RAJA to use a traversal template of the form:

```
template<typename LB>
void forall(omp_parallel_for , Index_type begin, Index_type end, LB body){
    #pragma omp parallel for
    for (int i = begin; i < end; ++i) body(i);
}
```

RAJA supports multiple ways to offload the execution on an accelerator. Considering GPU execution for example, a possible way is to use the OpenMP 4 accelerator model, which requires a definition of the form:

```
typedef RAJA::omp_parallel_for_acc exec_policy;
typedef RAJA::omp_acc_reduce       reduce_policy;
```

Such definition leads to a traversal template of the form:

```
template< typename LB >
void forall(omp_parallel_for_acc, Index_type begin, Index_type end, LB body){
    #pragma omp target
    #pragma omp parallel for
    for(int i = begin; i < end; ++i) body(i);
}
```

Note that the RAJA template cannot explicitly setup the GPU device data environment with an OpenMP `map` clause. The `map` clauses are used to specify how storage associated with specifically named variables is moved between host and device memories. Since a RAJA traversal is generic with respect to the loop body, it knows nothing about the data used in the loop. The OpenMP 4 standard fills the gaps to support ”unstructured data mapping“ that allows one to set up the proper device data environment before offloading via a RAJA traversal. We expect to manage such host-device data transfers in real application codes using a similar encapsulation approach to the way MPI communication is typically hidden.

For a CUDA-based execution, the notion of loops is absent, and execution should be mapped to a CUDA kernel, which is launched over a group of thread blocks on a CUDA-enabled GPU device. Each iteration executes on a different CUDA thread. To launch the loop as a CUDA kernel, the template parameters are:

```
typedef RAJA::cuda_acc    exec_policy;
typedef RAJA::cuda_reduce reduce_policy;
```

The following code snippets illustrate RAJA backend code for CUDA. So that the loop code continues to look like a loop, the loop body is passed to the traversal template (B), which has the same arguments as other traversals. This template launches a GPU kernel template (A) that executes each loop iteration on a separate GPU thread:

```
// (A) kernel template
template<typename LB>
__global__ void forall_cuda_kernel(Index_type begin, Index_type len , LB body){
    Index_type i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < len){
        body(begin+i) ;
    }
}

// (B) traversal template that launches CUDA GPU kernel
template<typename LB>
void forall( cuda_acc, int begin, int end, LB body){
    size_t blockSize = THREADS_PER_BLOCK;
    size_t gridSize  = (end - begin + blockSize - 1) / blockSize ;
    Index_type len = end - begin ;
    forall_cuda_kernel<<<gridSize , blockSize>>>(body , begin , len) ;
}
```

To manage data transfers between host and device, when using CUDA, we have multiple options. Using CUDA Unified Memory is the simplest and least intrusive method. Memory allocations are replaced with calls to `cudaMallocManaged()`, which allows data to be accessed in the same way on either the host or device with no explicit transfer operations. However, this may not yield desired performance in many situations. When this is the case, we can encapsulate CUDA memory copy routines in a manner similar to how we would use OpenMP unstructured data mapping.

IndexSets and Additional Traversal Features

Mesh-based multi-physics applications contain loops that iterate over mesh elements, and thus data arrays representing fields on a mesh, in a variety of ways. Some operations involve stride-1 array data access while others involve unstructured accesses using indirection arrays. Often, these different access patterns occur in the same physics operation. For code maintenance, such loop iterations are usually coded using indirection arrays since this makes the code flexible and relatively simple. In this section, some key features of RAJA IndexSets are described, along with their use to manage complex loop iteration patterns and address a variety of performance concerns. In particular, IndexSets provide a powerful mechanism to balance runtime iteration space definition with compile-time optimizations.

A RAJA IndexSet is an object that encapsulates a complete loop iteration space that is partitioned into a collection of *segments*, of the same or different segment types. Figure 4.38 shows two different types of simple `Segments`, a *range* and a *list* that may be used to iterate over different portions of an array. A RAJA `RangeSegment` object defines a contiguous set of iteration indexes with constraints applied to the iteration bounds and to the alignment of data arrays with memory constructs. For example, range Segments can be aligned multiples of the SIMD width or the SIMT width, to help compilers generate more efficient code. A RAJA `ListSegment` is a chunk of iterations that do not meet the range `Segment` criteria. It is important to note, that, with RAJA, we emphasize the tight association between a loop iteration and a *footprint* of data array elements in memory.

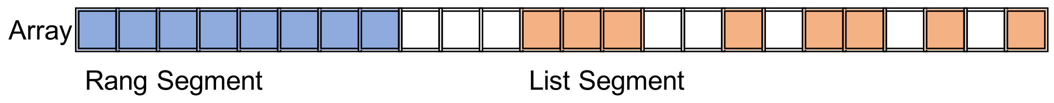


Figure 4.38: IndexSet segments in RAJA.

To illustrate some simple IndexSet mechanics, consider the following set of array indexes to process:

```
int num_elements = 21;
int elemenst[] = { 0 ,  1 ,  2 ,  3 ,  4 ,  5 ,  6 ,  7 , 14 ,  27 , 36 ,
                  40 , 41 , 42 , 43 , 44 , 45 , 46 , 47 , 87 , 117 };
```

Such a set of indexes may enumerate elements on a mesh containing a particular material in a multi-material simulation, for example. The indexes may be assembled at runtime into an `IndexSet` object by manually creating and adding `Segments` to the `IndexSet` object. A more powerful alternative is to use one of several parameterized *RAJA IndexSet builder* methods to partition an iteration space into a collection of ”work Segments“ according to some architecture-specific constraints. For example,

```
RAJA::Indexset segments = RAJA::createIndexset(elems, num elems);
```

might generate an `IndexSet` object containing two range Segments ($\{0, \cdots, 7\}$, $\{40, \cdots, 47\}$) and two list segments ($\{14, 27, 36\}$, $\{87, 117\}$).

When the `IndexSet` object is passed along with a loop body (lambda function) to a RAJA iteration template, the operation will be dispatched automatically to execute each of the Segments:

```
RAJA::forall<exec_policy>( Segments , [=] ( ... ) {
    /* loop body */
} );
```

That is, a specialized iteration template will be generated at compile-time for each Segment type. Iteration over the range Segments may involve a simple for-loop such as:

```
for(int i = begin; i < end; ++i) loop_body(i);
```

or iteration over the list Segments in a for-loop, with indirection applied:

```
for(int i = 0; i < seglen ; ++i) loop_body( Segment[i] );
```

IndexSet builder methods can be customized to tailor segments to hardware features and execution patterns to balance compile-time and runtime considerations. Presently, IndexSets enable a two-level hierarchy of scheduling and execution. A dispatch policy is applied to the collection of Segments. An execution policy is applied to the iterations within each segment. Examples include:

- Dispatch each segment to a CPU thread so segments run in parallel and execute range segments using SIMD vectorization.
- Dispatch segments sequentially and use OpenMP within each segment to execute iterations in parallel.
- Dispatch segments in parallel and launch each segment on either a CPU or a GPU as appropriate.

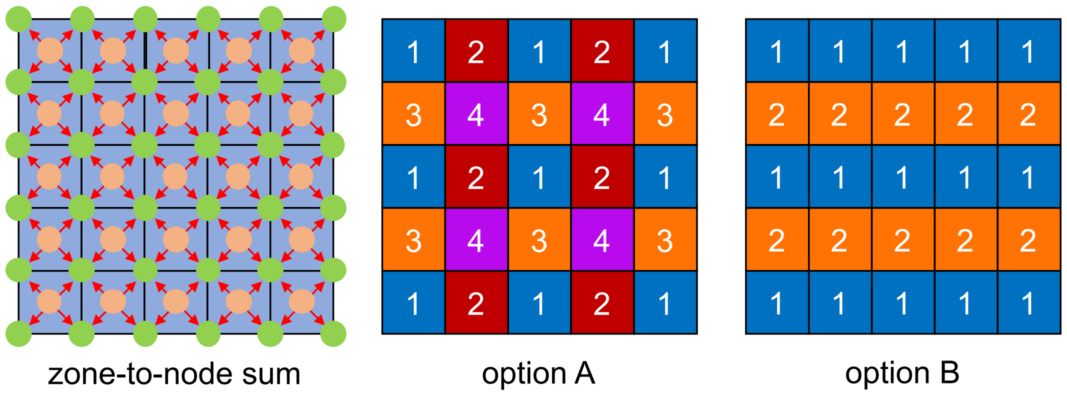


Figure 4.39: Zone-to-node sum with two loop ordering options in RAJA.

Loop Reordering and Tiling

RAJA `IndexSets` can expose available parallelism in loops that are not written using a parallel pattern. For example, a common operation in a staggered-mesh code, sums zonal values to surrounding nodes as is illustrated in the left image of Figure 4.39. IndexSets can be used to reorder loop iterations to achieve "data parallel" execution without modifying the loop body code. Figure 4.39 shows two different ordering possibilities, (A) and (B). Different colors indicate independent groups of computation, which can be represented as segments in indexSets. For option A, we iterate over groups (Segments) sequentially (group 1 completes, then group 2, etc.) and operations within a group (Segment) can be run in parallel. For option B, we process zones in each group (row) sequentially and dispatch rows of each color in parallel. It is worth emphasizing that no source code modifications are required to switch between these parallel iteration patterns. RAJA Segments can also represent arbitrary tilings of loop iterations that can be tuned and sized for specific architecture and memory configurations. When loop iterations are encapsulated in IndexSet Segments, data arrays can be permuted for better locality and cache reuse.

Kokkos vs. RAJA

Kokkos and RAJA are very similar in their objectives (both aiming at providing a performance portability layer for node level parallelism) and approaches (both focusing on abstracting away from serial loops; both are C++ templates/libraries/runtimes instead of language extensions, etc.). However there is one difference that is most relevant to the purpose of the SLATE project. Kokkos supports multi-dimensional arrays (Views) and allows customized data layout for the array, while RAJA only supports one-dimensional arrays with more support of random access. As multi-dimensional arrays are central to dense linear algebra, and the data layout of the array has a significant impact on performance, the flexibility of Kokkos arrays is better suited for SLATE purposes.

4.3 Consequences for SLATE

The ongoing evolution of the software technology, described in this chapter, creates unique opportunities for the development of the SLATE project. Here we summarize the most impactful aspects:

Software Engineering: While there is no question about the robustness of procedural languages, such as modern C and Fortran, for scientific computing, modern software engineering demands encapsulation, polymorphism, generic programming, etc., which are just so much easier to express in C++. It is only natural for SLATE to adopt C++ as its implementation language, to leverage its support for object oriented programming, as well as a plethora of other great features (exception handling, smart pointers, etc.).

Portability: As the OpenACC and OpenMP standards mature, and are adopted by the major compilers, there is less and less reasons to rely on proprietary solution for programming accelerators. While the actual implementations of the accelerator programming extensions may have some deficiencies for some time, solid foundations are already in place. There are no reasons to believe that OpenACC and/or OpenMP will not be able to fulfill SLATE's requirements for handling node-level memory consistency. On the other hand, portable frameworks, such as Kokkos and RAJA, should fulfill the need for custom kernel development, on rare occasions of gaps in the coverage of vendor libraries. Finally, MPI is the interface of choice for communication, and will be for the foreseeable future. At the same time, emerging programming frameworks, such as PaRSEC and Legion, are not to be dismissed, as a viable option for targeting exascale.

Platform Scalability: As SLATE will combine distributed memory programming with node-level programming and accelerator offload, it has the potential for becoming the solution of choice for all levels of the platform ladder: single node, multi-node, no GPU, single GPU, multi-GPU, Cloud instance, embedded system, mobile device.

CHAPTER 5

Matrix Layout Considerations

The BLAS standard defined the memory layout for matrices. In the original standard, a matrix is synonymous with a 2D Fortran array, meaning column-major storage. The CBLAS interface also supports 2D arrays stored in row-major, synonymous with the C language. Notably, the support for row-major does not require a separate implementation, but can be accomplished by switching the values of transposition and upper / lower parameters. One way or the other, the 2D array abstraction is a higher level language construct, which provides programming convenience, but causes highly suboptimal data access patterns when computing matrix multiplication. Historically, the significance of this fact used to be smaller, but becomes more prominent, as floating point capabilities of modern chips keep outpacing the memory systems. As a result, virtually all high performance BLAS implementations copy the input arrays to internal formats, in order to achieve maximum efficiency. This pays off in the case of large matrices, when the $O(N^2)$ cost of the copy is negligible compared to the $O(N^3)$ cost of the matrix multiplication. Traditionally, large matrices are the target of BLAS optimizations, since LAPACK and ScaLAPACK are structured to call large GEMM operations.

5.1 ATLAS Layouts

All the information about the inner workings of ATLAS, described in this section, was provided by the author of ATLAS, Cline Whaley.

From the standpoint of a BLAS implementation it is ideal if memory is accessed completely sequentially (consecutive memory locations). This results in the lowest possible bandwidth demand - best utilization of caches, highest benefit from prefetch operations, etc. The standard column-major or row-major format by no means allows for such access. The optimal access patter for the GEMM kernel is a function of loop optimizations and SIMD'zation. The main loop optimization technique used in ATLAS is *unroll and jam* with register blocking.

While the canonical form of matrix multiply looks like this:

```
for (i=0; i < M; i++)
   for (j=0; j < N; j++)
       for (k=0; k < K; k++)
          C(i,j) = C(i,j) + A(i,k) * B(k,j);
```

unrolled and jammed version looks like this:

```
for (i=0; i < M; i += 3)
   for (j=0; j < N; j += 2)
       for (k=0; k < K; k++)
       {
          C(i,j) += A(i,k) * B(k,j);
          C(i+1,j) += A(i+1,k) * B(k,j);
          C(i+2,j) += A(i+2,k) * B(k,j);
          C(i,j+1) += A(i,k) * B(k,j+1);
          C(i+1,j+1) += A(i+1,k) * B(k,j+1);
          C(i+2,j+1) += A(i+2,k) * B(k,j+1);
}
```

and like this with register blocking:

```
for (i=0; i < M; i += 3)
   for (j=0; j < N; j += 2)
   {
       register c00, c10, c20, c01, c11, c21;
       c00 = c10 = c20 = c01 = c11 = c21 = 0.0;
       for (k=0; k < K; k++)
       {
          register a0=A(i,k), a1=A(i+1,k), a2=A(i+2,k);
          register b0=B(k,j), b1=B(k,j+1);
          c00 += a0 * b0;
          c10 += a1 * b0;
          c20 += a2 * b0;
          c01 += a0 * b1;
          c02 += a1 * b1;
          c03 += a2 * b1;
       }
       C(i,j)     += c00;
       C(i+1,j)   += c10;
       C(i+2,j)   += c20;
       C(i,j+1)   += c01;
       C(i+1,j+1) += c11;
       C(i+2,j+1) += c21;
}
```

At the same time, virtually all modern CPUs get their performance from SIMD vectorization, which has further consequences for the data layout.

The term access-major layout was coined by Clint Whaley to describe an arrangement of matrix elements in memory corresponding to the access pattern of the GEMM implementation. I.e., the GEMM routine produces a consecutive memory access pattern during its execution. This has some profound consequences. First, one GEMM kernel is likely to have different storage patterns for each of its three operands. Second, two different GEMM kernels are likely to have incompatible storage patterns.

Currently, ATLAS is based on a new framework that supports several access-major storage patterns. The framework autogenerates routines that copy the data between the standard layout and the access-major layout, for the input arrays (A and B), and for the output array (C). Once the operands are in access-major storage, the ATLAS GEMM kernel always accesses them sequentially, generating the most efficient memory traffic.

The GEMM routine in ATLAS is based on a lower level building block called the *access-major matrix multiply Kernel* (ammmK). The dimensions of the ammmK are selected such that it fits in some level of cache. I.e., cache blocking is taken care of at a higher level, such that it is not a concern for the ammmK kernel itself. The focus of the ammmK kernel is instruction level parallelism, accomplished through loop optimizations, vectorization, register blocking, while implementing the access-major layout.

"The arrays used by ammmK have a more complex storage pattern, where the matrix has been permuted so that all arrays are naturally accessed in a purely sequential fashion when the computation is being performed. Completely sequential access allows us to minimize cache line conflicts, maximize cache line packing & hardware prefetch accuracy, and ensures that our bus access is as 'smooth' as possible (i.e. it minimizes the number of cache misses that happen at any one time)." – Clint Whaley

5.2 MKL Packed GEMM

Intel recently introduced MKL routines for multiplying matrices stored in a packed form, meaning internal, proprietary, opaque layout that is optimal for performance. Since the layout is opaque, MKL provides routines for allocating the space, translating the matrices, computing the multiplication, and freeing the space.

First, the space for copies of A and/or B needs to be allocated using one of the allocation functions (depending on precision):

```
float* cblas_sgemm_alloc (const CBLAS_IDENTIFIER identifier,
                          const MKL_INT m, const MKL_INT n, const MKL_INT k);

double* cblas_dgemm_alloc (const CBLAS_IDENTIFIER identifier,
                           const MKL_INT m, const MKL_INT n, const MKL_INT k);
```

Then the matrices can be packed using:

```
void cblas_sgemm_pack (const CBLAS_LAYOUT Layout, const CBLAS_IDENTIFIER identifier,
                       const CBLAS_TRANSPOSE trans,
                       const MKL_INT m, const MKL_INT n, const MKL_INT k,
                       const float alpha, const float *src, const MKL_INT ld,
                       float *dest);

void cblas_dgemm_pack (const CBLAS_LAYOUT Layout, const CBLAS_IDENTIFIER identifier,
                       const CBLAS_TRANSPOSE trans,
                       const MKL_INT m, const MKL_INT n, const MKL_INT k,
                       const double alpha, const double *src, const MKL_INT ld,
                       double *dest);
```

And then matrix multiplication can be performed using:

```
void cblas_sgemm_compute (const CBLAS_LAYOUT Layout,
                          const MKL_INT transa, const MKL_INT transb,
                          const MKL_INT m, const MKL_INT n, const MKL_INT k,
                          const float *a, const MKL_INT lda,
                          const float *b, const MKL_INT ldb,
                          const float beta, float *c, const MKL_INT ldc);

void cblas_dgemm_compute (const CBLAS_LAYOUT Layout,
                          const MKL_INT transa, const MKL_INT transb,
                          const MKL_INT m, const MKL_INT n, const MKL_INT k,
                          const double *a, const MKL_INT lda,
                          const double *b, const MKL_INT ldb,
                          const double beta, double *c, const MKL_INT ldc);
```

The packed copies can be free using:

```
void cblas_sgemm_free (float *dest);
void cblas_dgemm_free (double *dest);
```

The `identifier` parameter indicates if the operation (allocation, packing) applies to the A matrix or the B matrix (`CblasAMatrix` or `CblasBMatrix`).

The `Layout` parameter takes the standard CBLAS values (`CblasRowMajor` or `CblasColMajor`). However, it must use the same value for the entire sequence of related `cblas_?gemm_pack()` and `cblas_?gemm_compute()` calls. Also, for multithreaded calls, the same number of threads must be used for packing A and B. Intel also recommends that the same number of threads is used for packing and computing.

The `cblas_?gemm_compute` function can be called using any combination of packed or unpacked matrices A and B. The `transa` and `transb` parameters indicate if the corresponding matrix is packed. In addition to the standard CBLAS values (`CblasTrans`, `CblasNoTrans`, `CblasConjTrans`), they can also take the value `CblasPacked`, in which case the `lda` / `ldb` parameters are ignored.

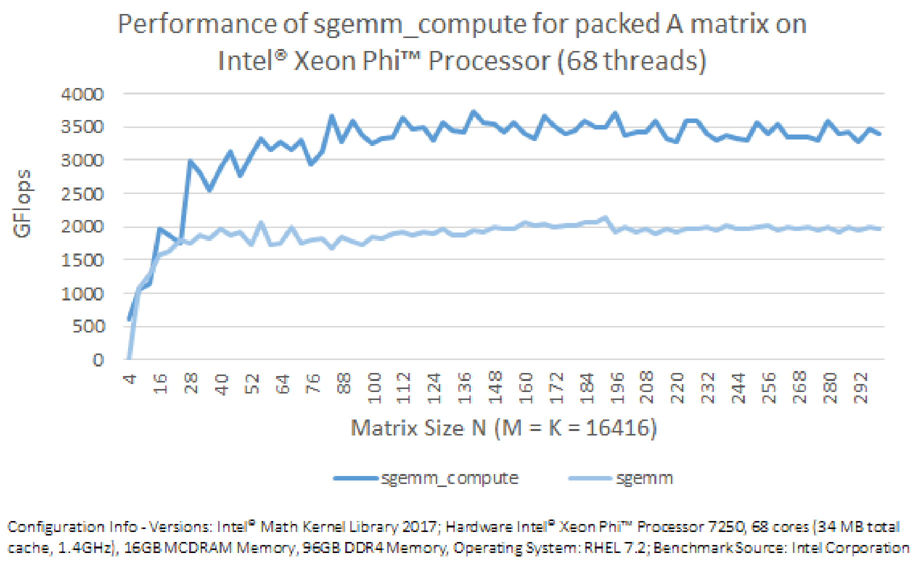


Figure 5.1: Performance of `sgemm_compute()` on Xeon Phi (source: https://software.intel.com/en-us/articles/introducing-the-new-packed-apis-for-gemm).

5.3 GPU Batched GEMM

Traditionally, dense matrices are stored in Fortran-style, column-major, layout. LAPACK relies on this layout, and ScaLAPACK relies on this layout to store the node-local portion of the matrix in the 2D block cyclic distribution. At the same time, there are numerous advantages of storing the matrix by tiles of relatively small size (128, 192, 256, ...). The PLASMA project stores matrices by tiles, with tiles arranged in a column-major layout and elements within tiles arranged in a column-major layout. The DPLASMA project relies on tile layout for storing node-local portions of distributed matrices.

Tile layout creates challenges for GPU acceleration. The standard GEMM routine cannot be used, as GPUs are not capable of executing them efficiently for small problems, one at a time. The solution is the use of batch GEMM operations, which execute a large number of small matrix multiplies concurrently. The question remains about the performance of batch operations compared to standard GEMMs, specifically the case of the Schur complement operations, critical to the performance of dense matrix factorizations.

Figure 5.2 shows the performance of the Schur complement operation using the NVIDIA Pascal GPU and the CUDA 8.0 SDK. The operation is $C = C - A \times B$, with C of size $40,000 \times 40,000$, A of size $40,000 \times k$, and B of size $k \times 40,000$. The darker curve shows the performance when the matrices are stored in the canonical, column-major layout, and the regular GEMM is used. The lighter curve shows the performance when the matrices are stored by tiles of size $k \times k$ and batched GEMM is used. Double precision is used in both cases.

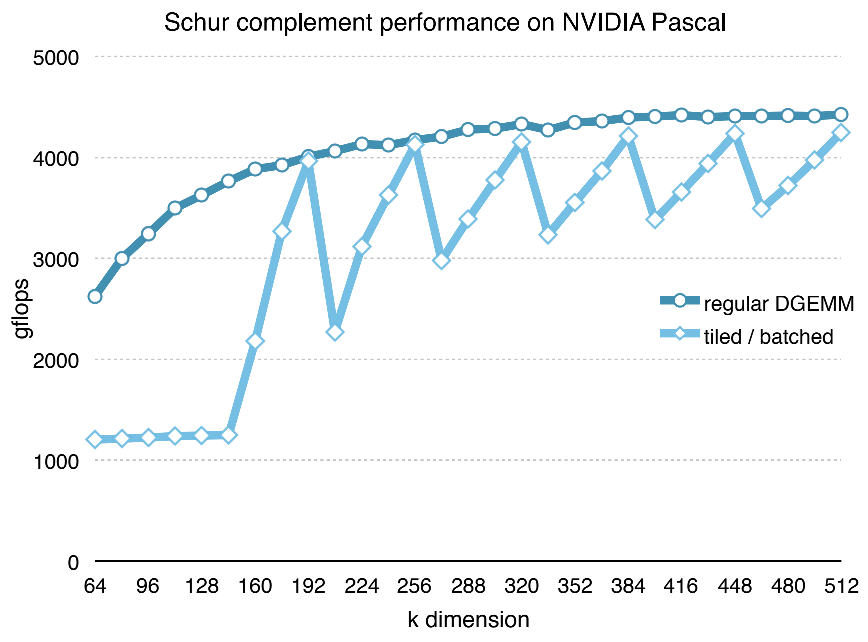


Figure 5.2: Performance of Schur complement on NVIDIA Pascal.

There are no surprises here. The standard GEMM curve is smooth, while the batched GEMM curve has a sawtooth shape, with peaks at the values of 192, 256, 320, 384, 448, 512, due to internal blocking of the GEMM implementations with the factor of 64. This is because very efficient code is used to handle 64-divisible regions, and much less efficient code is used to handle the remainder. In the case of the standard GEMM, the inefficiency only affects the outskirts of the large matrix. In the case of the batched GEMM, the inefficiency affects every tile.

The irregular pattern is by no means a problem for SLATE, as the tiling size may be chosen to match the tiling of the batched GEMM implementation. At the same time, it is not a fundamental challenge for GPU vendors to implement batched GEMM which matches the performance of the standard GEMM for certain tiling sizes. Therefore, the use of tile layout seems to be a legitimate choice for SLATE.

5.4 Consequences for SLATE

The observations of this chapter lead us to the following conclusions:

Presenting Opaque Matrix Layout makes perfect sense for SLATE, as this is the de facto modus operandi at the BLAS level anyways. At the same time, the transparency of the ScaLAPACK layout does not really provide much comfort in the face of its complexity.

Facilitating Packing is a great idea for SLATE, as two mainstream BLAS implementations, MKL and ATLAS, openly promote the solution, and the performance benefits of packing can be staggering (over 75% improvement reported by Intel).

Tiling is a legitimate option for SLATE, as GPUs are no longer handicapped by tile opeartions, due in large part to the ongoing standardization of batch BLAS routines, and their availability in vendor supplied libraries.

CHAPTER 6

Algorithmic Considerations

6.1 LU

Solving a dense, non-symmetric, system of linear equations is a fundamental dense linear algebra capability, and the LU factorization is usually the method of choice, because of its practical numerical stability and low operation count, compared to more robust methods. It is commonly know as the operation behind the MATLAB backslash operator, and as the benchmark of the TOP500 list. Because of its role as a benchmark, it is usually one of the first workloads implemented and optimized for new architectures.

The Netlib implementation of the *High Performance LINPACK* (HPL) benchmark [37] is a testimony to the optimization efforts required for maximum performance of the LU factorization on distributed memory systems, especially when compared to the implementation of the LU factorization routine in ScaLAPACK, which could be taken as the baseline here. One thing that the HPL inherited from ScaLAPACK is the 2D block cyclic data distribution. Otherwise, the Netlib HPL is basically a custom implementation.

Most optimization effort goes into dealing with the bottleneck of the panel factorization, which is inherently inefficient and lies on the critical path of the algorithm. To ensure fast execution of the panel factorization, HPL moved from an iterative approach to a recursive approach, and actually provides a couple of different recursive implementations. It also relies on a few different custom implementations of the collective communication operations, in order to be able to overlap communication with computation, as at the time of writing the code MPI did not support non-blocking collectives. Finally, HPL utilizes the idea of lookahead, which allows to overlap the panel factorization with the update of the trailing submatrix, so that the inefficient operations run in the background of the efficient matrix multiplication. At the same time, the Netlib HPL is an outdated code, as it has no notion of multicore CPUs or GPUs, and is usually heavily modified before being used to benchmark modern machines.

There have also been numerous attempts at improving the algorithms in order to remove the panel factorization bottleneck. In the 2014 survey, we analyzed the impact of different approaches on the performance and numerics of the algorithm [31]. Specifically, we looked at the effects of: incremental pivoting, tournament pivoting, and applying random butterfly transformation to the input matrix instead of pivoting. When the numerical accuracy suffered, we tried to recover it with iterative refinement. The consensus of that work was, more less, that the traditional algorithm with partial pivoting, works best if implemented well. This study was done in the context of multicore CPUs only - no GPUs, no distributed memory.

An important contribution to the topic is the idea of *Parallel Cache Assignment* (PCA) introduced by Castaldo and Whaley [25]. It relies on the observation that BLAS 2 operations are efficient if executed in cache. For maximum performance on multicores, the panel factorization is multithreaded, with static assignment of chunks of the panel to threads / cores. At the same time, synchronization is done using the memory consistency protocol implemented in hardware, instead of slower software mechanisms, such as mutexes.

Subsequently, we applied the idea of cache residency to a recursive implementation [32, 34], which became the basis for the panel factorization in the PLASMA library. This implementation delivered performance way in excess of the memory barrier and scaled very well with the number of cores. However, a short study was conducted before porting the PLASMA library from the QUARK runtime to the OpenMP runtime, which showed inferior performance of plain recursion compared to simple blocking. Therefore, the current PLASMA implementation is based on cache residency and low level synchronization, but blocking rather than recursion. Precise performance experiments for the new panel factorization routine have not been conducted yet. The added benefit of the blocked implementation is its reduced complexity.

Given the level of trust that the technical computing community has for the LU factorization with partial pivoting, this algorithm is the most likely target for SLATE implementation. Ideally, the SLATE implementation of the LU factorization and solve should serve as a replacement for the HPL benchmark for the CORAL machines and the exascale machines to follow. Critical to accomplishing this objective is a fast implementation of the panel factorization and application of the lookahead technique.

6.2 LDLT

Many applications require the solution of dense linear systems of equations, whose coefficient matrices are symmetric indefinite. To solve such linear systems on a shared-memory computer, LAPACK computes the LDL^T factorization of the matrix using the Bunch Kaufman [22] or rook pivoting [9], or the LTL^T factorization using the Aasen's algorithm [6, 71]. The main challenge is that in order to maintain the numerical stability of the factorization, the algorithms require symmetric pivoting. This symmetric pivoting leads to the data accesses and dependencies that make it difficult to obtain high performance of the factorization. For instance, with the symmetric pivoting, it becomes a challenge to integrate the lookahead, which was a critical component in obtaining the high performance of the LU factorization (Section 6.1). ScaLAPACK still does not support a symmetric indefinite solver.

Even with the performance challenges, the symmetric indefinite factorization has several numerical and structural advantages over the nonsymmetric factorization. For instance, under the symmetric factorization, the matrix inertia stays the same. Moreover, when factorizing a diagonal block of a symmetric matrix, it is critical to maintain the symmetry in order to maintain the computational and storage costs of the overall symmetric factorization.

To improve the performance of the LTL^T factorization, a communication-avoiding (CA) variant of the Aasen's algorithm has been proposed [12]. The algorithm first reduces the symmetric matrix into a band form based on the tiled Aasen's algorithm. At each step, the algorithm factorizes a block column, or panel, by first updating the panel in a left-looking fashion and then LU factorizing the panel. Finally, in the second stage of the algorithm, the band matrix is factorized. The advantage of this algorithm is that the first stage dominates the computational cost of the whole factorization process, and for this first stage, we can utilize the optimized LU panel factorization (Section 6.1), while the rest of the computation is mostly based on BLAS-3.

We have studied the performance of the CA Aasen's algorithm with PLASMA [7]. Compared with the right-looking update of the symmetric indefinite factorization in LAPACK, the CA Aasen's left-looking update has a limited parallelism. To increase the parallelism for updating each tile of the panel, PLASMA applies a parallel reduction and accumulate a set of independent updates into a user-supplied workspace. How much parallelism the algorithm can exploit depends on the number of tiles in the panel and the amount of the workspace provided by the user. Then, the panel is factorized using the PLASMA's multi-threaded LU panel factorization routine. Finally, we use the PLASMA's band LU factorization routine for the second stage of the factorization. Since there is no explicit global synchronization between the stages, a task to factorize the band matrix can be started as soon as all the data dependencies are satisfied. This allows the execution of these two algorithms to be merged, improving the parallel performance, especially since both algorithms have limitted amount of parallelism that can be exploited. Our performance studies have demonstrated that especially on a manycore architecture, the CA Aasen's algorithm, combined with the runtime, can obtain significant speedups over the threaded MKL.

6.3 QR/LQ

The QR factorization is a fail-safe method of solving linear systems of equations and the method of choice for solving linear least squares problems. While QR is rarely an attractive option for solving linear systems of equations, where usually the practical stability of the cheaper LU factorization suffices, it is the main option for solving least squares problems. Here, however, it faces the challenge of dealing with highly overdetermined systems, resulting in very tall and thin matrices, usually referred to as *tall and skinny*.

The basic problem is that a thin matrix does not expose much parallelism if the traditional algorithms is applied, which eliminates one full column of the matrix at a time. To address this issue, classes of algorithms were developed referred to as *Tall and Skinny QR* (TSQR) and *Communication Avoiding QR* (CAQR). Seminal work in this area was done by Demmel et al. [28]. The basic idea is that the panel is split vertically, into shorter subpanels, which can be reduced in parallel. The initial parallel reduction leaves unreduced elements (R factors from each subpanel reduction), which can them be pairwise reduced in a tree-like pattern. This approach has tremendous performance benefits for factoring tall matrices in parallel.

This idea was taken to extremes in the PLASMA project producing the class of *tile algorithms* [23, 24]. In the basic tile QR factorization, the panel is reduced incrementally, one square tile at a time. This allows for simultaneously applying updates to the trailing submatrix, resulting in perfect pipelining and producing outstanding *strong scaling*. On the other hand, a naive implementation leads to a staggering 50% overhead in floating point operations. The remedy is internal blocking of the algorithm by a factor $IB << NB$, where NB is the tile size. While this reduces the extra operations, it moves the algorithm away from being compute intensive and closer to being memory intensive.

Interesting work has been done on analyzing different reduction patterns in order to minimize the length of the critical path [33]. This led to the implementation of a few different patterns in the PLASMA library, along with a mechanism for easy generation and application of the Q matrix for arbitrary reduction patterns.

However, the real Achilles heal of the tile QR factorization is the complexity of the required kernels, specifically the fact that the update of the trailing submatrix is not a simple matrix multiplication, but a series of smaller matrix multiplications. This exposes the kernel to multiple overheads. One source of overheads is the fact that most BLAS implementations (MKL, ATLAS) copy the input matrices to a performance-oriented layout before the actual operation. Another is the overhead of invoking *cleanup code* when the input is not divisible by the internal blocking factors. Finally, while GEMM-based updates can be implemented easily on GPUs using the batched interface, the tile QR updates require custom kernels, which can rarely match the performance of vendor provided GEMM.

While TSQR/CAQR algorithms solidified their credibility in the dense matrix community, the tile algorithms have not really gained that much traction and, due to their demand for custom kernels, failed to penetrate the GPU computing field. Therefore, it seems to be the best choice for SLATE to utilize the TSQR/CAQR class of algorithms, but not go as far as tile QR algorithms. This will require a fast parallel implementation of a LAPACK-style QR panel factorization, but will provide the performance benefit for using simple GEMM calls for updating the trailing submatrix.

6.4 Mixed Precision

On modern architectures, single precision 32-bit floating point arithmetic (FP32) is usually twice as fast as double precision 64-bit floating point arithmetic (FP64). The reason for this is that the amount of bytes moved through the memory system is essentially halved and the circuit logic inside the floating-point units (FPUs) allows to double the execution rate for twice as short data types. Indeed, on most current multicore CPUs, high-end AMD GPUs (e.g., FirePro W9100), Intel Xeon Phi, and NVIDIA Pascal GPUs, the single precision peak is twice the double precision peak. On most high-end NVIDIA GPUs (e.g., the GeForce GTX Titan Black and server Kepler cards) the ratio of single precision peak vs. double precision peak is 3-fold, but can go up to $32\times$ (e.g., on the Titan X) depending on the ratio of the available 32-bit to the 64-bit CUDA cores.

6.4.1 Linear Systems

A common approach to the solution of dense linear systems is to perform the LU factorization of the coefficient matrix using Gaussian elimination. First, the coefficient matrix A is factored into the product of a lower triangular matrix L and an upper triangular matrix U. Partial row pivoting is used to improve numerical stability resulting in a factorization $PA = LU$, where P is a permutation matrix. The solution for the system is achieved by first solving $Ly = Pb$ (*forward substitution*) and then solving $Ux = y$ (*backward substitution*). Due to round-off errors, the computed solution x carries a numerical error magnified by the condition number $\kappa(A)$ of the coefficient matrix A.

In order to improve the computed solution, we can apply an iterative process which produces a correction to the computed solution at each iteration, which then yields the method that is commonly known as the *iterative refinement* algorithm. As Demmel points out [29], the non-linearity of the round-off errors makes the iterative refinement process equivalent to the Newton's method applied to the function $f(x) = b - Ax$. Provided that the system is not too ill-conditioned, the algorithm produces a solution correct for the working precision. Iterative refinement in double/double precision is a fairly well understood concept and was analyzed by Wilkinson [84], Moler [60], and Stewart [78].

The algorithm can be modified to use a mixed precision approach. The factorization $PA = LU$, the solution of the triangular systems $Ly = Pb$, and $Ux = y$ are computed using single precision arithmetic. The residual calculation and the update of the solution are computed using double precision arithmetic and the original double precision coefficients. The most computationally expensive operation, the factorization of the coefficient matrix A, is performed using single precision arithmetic and takes advantage of its higher speed. The only operations that must be executed in double precision are the residual calculation and the update of the solution. The only operation with computational complexity of $\mathcal{O}(n^3)$ is handled in single precision, while all operations performed in double precision are of at most $\mathcal{O}(n^2)$ complexity.

The only drawpack is the memory overhead. The coefficient matrix A is converted to single precision for the LU factorization and the resulting factors are stored in single precision while the initial coefficient matrix A needs to be kept in memory. Therefore, the algorithm requires 50% more memory than the standard double precision algorithm.

Currently, mixed precision iterative refinement linear systems solvers are implemented in LAPACK, as well as the PLASMA and MAGMA libraries. On standard multicore CPUs, iterative refinement typically delivers speedups between 1.6× and 1.8×. The impact of iterative refinement is the highest for artchitectures, with seriously handicapped double precision performance. The historical reference is the Cell processor, where double precision arithmetic was 14× slower, resulting in 7× speedup [56, 57] The technique is also used in the MAGMA library for GPUs and multi-GPU systems [81]. Recent experiments on the Titan X GPU, where double precision is 32× slower, produced 26× speedups.

6.4.2 Other Algorithms

We have shown how to derive the mixed precision versions of a variety of algorithms for solving general linear systems of equations. In the context of overdetermined least squares problems, the iterative refinement technique can be applied to either the augmented system, where both the solution and the residual are refined [27], or to the QR factorization, or to the semi-normal equations, or to the normal equations [19]. Iterative refinement can also be applied to eigenvalue computations [36] and to singular value computations [35].

Recently, we developed an innovative mixed-precision QR for tall-and-skinny matrices [86] that uses higher-precision at critical parts of the algorithm, resulting in increased numerical stability and several times speedup over the standard algorithms (like CGS, MGS, or Householder QR factorizations). In particular, the algorithm starts from a Cholesky QR algorithm, which is known to be fast (expressed as Level 3 BLAS) but numerically unstable, as the computation goes through normal equations. However, computing the normal equations and other critical parts of the algorithm in double-double precision is shown to be stable, while preserving the performance profile for Level-3 BLAS operations [87].

6.5 Matrix Inversion

Matrix inversion is not an appropriate method for solving a linear system of equations. The appropriate method is matrix factorization, such as LU or Cholesky, followed by forward and backward substitution. At the same time, multiple applications require the computation of the actual inverse. A canonical example is the computation of the variance-covariance matrix in statistics. Higham lists more such applications [51]. The need for computing the inverse was also expressed by some of the ECP apps teams (computational chemistry, material science).

Computing the matrix inversion has been an attractive target for research because of its optimization opportunities [8, 17], the Cholesky-based inversion more so than the LU-based inversion. State of the art implementation of the Cholesky inversion is implemented in the PLASMA library. Top performance is achieved by removal of anti-dependencies, careful ordering of loops, and pipelining of all the stages of the algorithm.

Anti-Dependencies Removal: LAPACK and ScaLAPACK take a very conservative approach to memory management. As a result, all stages of the matrix inversion are performed in place, as the input matrix is gradually overwritten by the output result. From the standpoint of work scheduling, this creates a lot of anti-dependencies, which prevent efficient execution. This is remedied by allocating temporary storage and performing operations out of place. In the case of matrix inversion all anti-dependencies can be removed this way.

Optimal Loop Ordering: The three stages constituting the matrix inversion: factorization (POTRF), triangular inversion (TRTRI) and triangular matrix multiplication (LAUUM), all contain a large number of GEMM operations, which are commutative. At the same time, their ordering heavily impacts the length of the critical path. The shortest critical path, and the maximum parallelism, is achieved through the correct ordering. The work by Agullo contains detailed analysis [8].

Complete Pipelining of Stages: Finally, in the case of LAPACK and ScaLAPAC, the three stages (POTRF, TRTRI, LAUUM) are executed in a sequence, one at a time, and each one is affected by the load imbalance towards the end of the execution. A superior approach is to form a single task graph, encompassing all three stages, and schedule all tasks, based on their data dependencies. This leads to a very high degree of pipelining between the stages and superior performance.

6.6 Eigenvalue and Singular Value Problems

Eigen decomposition is a fundamental workload of dense linear algebra, with critical importance to structural engineering, quantum mechanics, and many other areas of technical and scientific computing. So is the singular value decomposition (SVD) with applications in principal component analysis, digital image processing, information retrieval systems, to just name a few.

The eigenvalue problem is the problem of finding an eigenvector x and eigenvalue λ that satisfy $Ax = \lambda x$, where A is a symmetric or nonsymmetric $n \times n$ matrix. Eigendecompositin of a matrix is a decomposition of the for $A = X \Lambda X^{-1}$, where Λ is a diagonal matrix of eigenvalues and X is a matrix of eigenvectors.

The objective of the singular value decomposition is to find orthogonal matrices U and V, and a diagonal matrix Σ with nonnegative elements, such that $A = U \Sigma V^T$, where A is an $m \times n$ matrix. The diagonal elements of Σ are the singular values of A, while the columns of U and V are its left and right singular vectors, respectively.

Typically, solutions to singular value problems and the eigenvalue problems are found by following the similar three stage process:

Reduction: Orthogonal transformations are applied to the input matrix from the left and from the right to reduce it to a condensed form (bidiagonal for SVD, tridiagonal for symmetric eigendecomposition, and Hessenberg for non symmetric eigendecomposition).

Solution : An iterative solver is applied to further condense the matrix in order to find its eigenvalues or singular values.

Vector Computation: If desired, the eingenvectors or singular vectors are computed, by first finding the eigen/singular vectors of the condensed matrix and then finding the eigen/singular vectors of the original matrix in the process of back-transformation.

6.6.1 Singular Value Decomposition

For the singular value decomposition (SVD), two orthogonal matrices Q and P are applied on the left and right side of A, respectively, to reduce A to bidiagonal form, $B = Q^T AP$. Divide and conquer or QR iteration is then used as a solver to find both the singular values and the left and the right singular vectors of B as $B = \widetilde{U} \Sigma \widetilde{V}^T$, yielding the singular values of A. If desired, singular vectors of B are back-transformed to singular vectors of A as $U = Q\widetilde{U}$ and $V^T = P^T \widetilde{V}^T$. In this section we describe, in detail, the three computational phases involved in the Singular Values Decomposition.

Classic Reduction to Bidiagonal Form

Due to its high computational complexity of $O(\frac{8}{3}n^3)$ (for square matrices) and interdependent data access patterns, the bidiagonal reduction phase is the most challenging phase. In the classic approach of LAPACK, referred to as the "one-stage algorithm", orthogonal transformations are used reduce the dense matrix to the bidiagonal form in one sweep. Performance of this algorithms is capped by the memory-bound Level 2 BLAS gemv routine.

In the case when all singular vectors are computed, reduction to the bidiagonal form requires more than 70% of all computational time. When only singular values are needed, the reduction requires about 90% of the total time. Because of the inefficiency of the

the classic approach, a new technique has been developed, referred to as the "two-stage" algorithms [18, 45–47, 58, 59] Im the two-stage algorithm, the matrix is first reduced to a band form, and then reduced to the "proper" bidiagonal form in the process of band reduction.

Two Stage Reduction to Bidiagonal Form

The two-stage reduction is designed to overcome the limitations of the one-stage reduction, which relies heavily on memory-bound operations. The algorithm is split into the first stage, which reduces the original matrix to a band matrix, and the second stage, which band matrix to the canonical bidiagonal form.

The computational cost of the first stage is $\sim \frac{8}{3}n^3$ floating point operations. This stage is compute bound and has a high degree of parallelism. Therefore, it can be implemented very efficiently. The second stage is much less compute intensive and has much lower degree of parallelism, but is also responsible for a much smaller amount of overall operations. Also, it can be implemented in a cache friendly manner, colloquially referred to as Level 2.5 BLAS [45, 47].

Bidiagonal Singular Solver

A bidiagonal singular solver computes the spectral decomposition of a bidiagonal matrix B such that $B = \widetilde{U}\Sigma\widetilde{V}^H$, with $\widetilde{U}\widetilde{U}^H = I$ and $\widetilde{V}\widetilde{V}^H = I$, where $\widetilde{U}$ and $\widetilde{V}^H$ are the singular vectors, and Σ are the singular values of B. The solution is usually found either using the QR algorithm [42], or the divide and conquer algorithm [44, 55].

Singular Vector Computation

In the case of the two-stage approach, the first stage reduces the original dense matrix A to a band matrix A_{band} such that $Q_1^H A P_1 = A_{band}$. Similarly, the second stage reduces the band matrix A_{band} to the bidiagonal form such that $Q_2^H A_{band} P_2 = B$. Consequently, the singular vectors are computed be multiplied by both Q_* and P_*, according to the formula:

$$\begin{aligned} U &= Q_1 Q_2 \widetilde{U} = (I - G_1 T_1 G_1^H)(I - G_2 T_2 G_2^H)\widetilde{U}, \\ V^H &= \widetilde{V}^H P_2^H P_1^H = \widetilde{V}^T (I - W_2 Tr_2^H W_2^H)(I - W_1 Tr_1^H W_1^H), \end{aligned}$$

where (G_1, T_1 and W_1, Tr_1) and (G_2, T_2 and W_2, Tr_2) represent the left and the right Householder reflectors generated during the first and the second stages of the reduction to the bidiagonal form. It is clear that the two-stage approach introduces a non-trivial amount of extra computation - the application of Q_2 and P_2^H - for the case when the singular vectors are needed.

Experiments showed that the two-stage algorithm can reach between $2\times$ and $3\times$ speedup when both the left and the right singular vectors are computed. At the same time, when only the singular values are needed, the two-stage approach can reach more than $7\times$ speedup [49].

6.6.2 Symmetric Eigenvalue Problem

Reduction to Tridiagonal Form

Wile singular value decomposition requires reduction to the bidiagonal form, symmetric eigendecomposition requires reduction to the tridiagonal form. Similarly to the singular value decomposition, the fast tridiagonal reduction algorithm is based on a two stage reduction. In the first stage the full symmetric matrix is reduced to a band symmetric matrix ($A \longrightarrow A_{band}$), and in the second stage, the band matrix is reduced to the tridiagonal matrix, in a process very similar to the one used for SVD [48, 50, 75].

Tridiagonal Eigensolver

A tridiagonal eigensolver is used to compute eigenpairs of the tridiagonal matrix, $T = Z\Lambda Z^T$, where Z is the matrix of orthogonal eigenvectors of T, and Λ is the diagonal matrix of eigenvalues. Four algorithms are available: QR iterations, Bisection and Inverse Iteration (BI), Divide and Conquer (D&C), and Multiple Relatively Robust Representations (MRRR). Discussion of the first two algorithms can be found in the book by Demmel [29]. A performance comparison of different symmetric tridiagonal solvers, by Demmel et al. [30], shows that the D&C and the MRRR solvers are the fastest available.

While D&C requires a larger extra workspace, MRRR is less accurate. Accuracy is a fundamental parameter, because the tridiagonal eigensolver is known to be the part of the overall symmetric eigensolver where accuracy can be lost. D&C is more robust than MRRR, which can fail to provide an accurate solution in some cases. In theory, MRRR is a $O(n^2)$ algorithm, whereas D&C is between $O(n^2)$ and $O(n^3)$, depending on the matrix properties. In many real-life applications, D&C is often less than cubic while MRRR seems to be slower than expected due to the number of floating point divisions and the cost of the iterative process. The main advantage of MRRR is that computation of a subset of eigenpairs is possible, reducing the complexity to $O(nk)$ for computing k eigenpairs.

Eigenvector Computation

After the reduction to condensed form, the eigensolver finds the eigenvalues Λ and eigenvectors Z of T. The eigenvalues are the same as for the original matrix A. To find the eigenvectors of the original matrix A, the eigenvectors Z of T need to be back-transformed by applying the same orthogonal matrices, Q_1 and Q_2, that were used in the reduction to the condensed form. This step is a series of `DGEMM` operations, usually achieves a high fraction of the machine's peak performance, and ends up being a small percentage of the total execution time.

The two-stage approach to the eigendecomposition has similar performance characteristics as the two-stage approach to the singular value decomposition, and delivers up to $3\times$ speedup when all the eigenvectors are computed, and up to $7\times$ speedup when only the eigenvalues are computed.

6.6.3 Nonsymmetric Eigenvalue Problem

The nonsymmetric eigenvalue problem is to find the scalar λ and the vector x satisfying $Ax = \lambda x$, when A is a nonsymmetric $n \times n$ matrix. In addition to this *left eigenvector* x, there is also the *right eigenvector* y, such that $y^T A = \lambda y^T$. In the symmetric case, left and right eigenvectors are identical.

Similarly to the symmetric case, the solution to the nonsymmetric eigendecomposition consists of three phases [43]. First, the matrix is reduced to the upper Hessenberg form by applying orthogonal transformations to form $H = Q_1^T A Q_1$. Then QR iteration is applied, which reduces the Hessenberg matrix to the upper triangular Schur form, $S = Q_2^T H Q_2$, revealing the eigenvalues of A as the diagonal elements of S. Finally, the eigenvectors Z of the Schur form S are computed, and transformed to the eigenvectors X of the original matrix A in the process of back-transformation.

Unfortunately, the two-stage approach is not easily applicable to the Hessenberg reduction. While a full matrix can be efficiently reduced to the band Hessenberg form [16], there is no efficient process for the band reduction. The only consolation is that the traditional Hessenberg reduction can easily be offloaded to GPUs to take advantage of their high memory bandwidth [80]. Recent developments in non symmetric eigenvalue solvers also include improvement of the eigenvector calculations by using Level 3 BLAS operations in the step of back transformation [41].

6.7 Consequences for SLATE

The experiences of the past decade of algorithmic developments in dense linear algebra indicate that SLATE should:

- Implement the canonical LU factorization with partial (row) pivoting. This is the most trusted linear solver algorithm of the technical computing community, and performs well when implemented well. Ideally, the SLATE implementation of the LU factorization and solve should be hard to beat by a hand optimized implementation of the HPL benchmark.
- Implement the LDLT factorization based on the communication avoiding variant of the Aasen's algorithms. The algorithm provides undeniable performance benefits, without substantially worsening numerical stability.
- Implement the communication-avoiding variants of the QR/LQ factorizations, but stop short of implementing the tile QR/LQ algorithms. While the communication avoiding features provide massive performance boost for tall and skinny matrices, the tile rendition shows performance limitations on CPUs and demands custom kernels on GPUs.

- Equip all its linear solvers with mixed precision capabilities, as the technique is generally beneficial on current hardware, if numerical properties of the input matrices grant its usage.
- Exploit pipelining potential of the matrix inversion routines for the performance boost and improved *strong scaling* properties.
- Implement singular value routines and eigenvalue routines based on the two-stage approach, which is clearly superior to the traditional algorithms on current hardware.

CHAPTER 7

Conclusions

This chapter contains a brief summary of all the findings of the previous chapters. More detailed conclusions can be found in the sections titled “Consequences for SLATE” included at the end of each chapter. The main findings of this document are:

SLATE is essential to the success of a large number of ECP applications, as a modern replacement for LAPACK and ScaLAPACK.

SLATE needs to target powerful nodes, with large numbers of cores, and powerful accelerators. This implies the use of breath-first (right-looking) algorithms, which produce large amounts of parallel work at a time, and the use of batch operations.

SLATE needs to cope with bandwidth limitations, which calls for very conservative use of the network. This implies heavy reliance on collective communication, preferably non-blocking collectives.

SLATE needs to target complex and deep memory systems. This implies the need for alternatives to the 2D block cyclic matrix layout of ScaLAPACK. SLATE needs to offer a much higher level of flexibility in laying out the matrix in the memory.

SLATE needs to leverage the tremendous progress in software technology, that took place since the introduction of ScaLAPACK. This includes new programing models of OpenMP and OpenACC, as well as major improvements to the basic distributed programming model of MPI. This also includes emerging technologies, such as node-level programing solutions, like Kokkos and RAJA, and distributed tasking systems, like PaRSEC and Legion.

SLATE needs to take advantage of the tremendous progress in dense linear algebra algorithms, made in the last decade, as major improvements were made to most of the algorithms of ScaLAPACK.

Bibliography

[1] Whitepaper: NVIDIA Tesla P100, . URL https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[2] Inside Volta: The World's Most Advanced Data Center GPU, . URL https://devblogs.nvidia.com/parallelforall/inside-volta/.

[3] Workshop on portability among HPC architectures for scientific applications. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis*, November 2015. URL http://hpcport.alcf.anl.gov/.

[4] DOE centers of excellence performance portability meeting, 2016. URL https://asc.llnl.gov/DOE-COE-Mtg-2016/.

[5] Oak Ridge Leadership Computing Facility - Summit, 2017. URL https://www.olcf.ornl.gov/summit/.

[6] Jan Ole Aasen. On the reduction of a symmetric matrix to tridiagonal form. *BIT Numerical Mathematics*, 11(3):233–242, 1971.

[7] Maksims Abalenkovs, Negin Bagherpour, Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Samuel Relton, Jakub Sistek, David Stevens, Panruo Wu, Ichitaro Yamazaki, Asim YarKhan, and Mawussi Zounon. PLASMA 17 performance report: Linear systems and least squares Haswell, Knights Landing, POWER8. Technical Report UT-EECS-17-750, University of Tennessee, 2017. URL http://www.netlib.org/lapack/lawnspdf/lawn292.pdf.

[8] Emmanuel Agullo, Henricus Bouwmeester, Jack Dongarra, Jakub Kurzak, Julien Langou, and Lee Rosenberg. Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. In *International Conference on High Performance Computing for Computational Science*, pages 129–138. Springer, 2010.

[9] Cleve Ashcraft, Roger G Grimes, and John G Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM Journal on Matrix Analysis and Applications*, 20(2):513–561, 1998.

[10] Guillaume Aupy, Mathieu Faverge, Yves Robert, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. *Euro-Par 2013: Parallel Processing Workshops: BigDataCloud, DIHC, FedICI, HeteroPar, HiBB, LSDVE, MHPC, OMHI, PADABS, PROPER, Resilience, ROME, and UCHPC 2013, Aachen, Germany, August 26-27, 2013. Revised Selected Papers*, chapter Implementing a Systolic Algorithm for QR Factorization on Multicore Clusters with PaRSEC, pages 657–667. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[11] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[12] Grey Ballard, Dulceneia Becker, James Demmel, Jack Dongarra, Alex Druinsky, Inon Peled, Oded Schwartz, Sivan Toledo, and Ichitaro Yamazaki. Communication-avoiding symmetric-indefinite factorization. *SIAM Journal on Matrix Analysis and Applications*, 35 (4):1364–1406, 2014.

[13] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[14] Michael Edward Bauer. *Legion: Programming distributed heterogeneous architectures with logical regions*. PhD thesis, Stanford University, 2014.

[15] J Bennett, R Clay, et al. ASC ATDM level 2 milestone #5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms. Technical Report SAND2015-8312, Sandia National Laboratories, 2015. URL http://www.sci.utah.edu/publications/Ben2015c/ATDM-AMT-L2-Final-SAND2015-8312.pdf.

[16] Michael W Berry, Jack J Dongarra, and Youngbae Kim. A parallel algorithm for the reduction of a nonsymmetric matrix to block upper-Hessenberg form. *Parallel Computing*, 21(8):1189–1211, 1995.

[17] Paolo Bientinesi, Brian Gunter, and Robert A Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Transactions on Mathematical Software (TOMS)*, 35(1):3, 2008.

[18] Paolo Bientinesi, Francisco Igual, Daniel Kressner, and Enrique Quintana-Ortí. Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures. *Parallel Processing and Applied Mathematics*, pages 387–395, 2010.

[19] ˚Ake Björck. *Numerical methods for least squares problems*. SIAM, 1996.

[20] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, et al. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1432–1441. IEEE, 2011.

[21] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1):37–51, 2012.

[22] James R Bunch and Linda Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of computation*, pages 163–179, 1977.

[23] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.

[24] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1): 38–53, 2009.

[25] Anthony M Castaldo and R Clint Whaley. Scaling LAPACK panel operations using parallel cache assignment. In *ACM Sigplan Notices*, volume 45, pages 223–232. ACM, 2010.

[26] Michel Cosnard, Emmanuel Jeannot, and Tao Yang. Compact DAG representation and its symbolic scheduling. *Journal of Parallel and Distributed Computing*, 64(8):921–935, 2004.

[27] James Demmel, Yozo Hida, E Jason Riedy, and Xiaoye S Li. Extra-precise iterative refinement for overdetermined least squares problems. *ACM Transactions on Mathematical Software (TOMS)*, 35(4):28, 2009.

[28] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[29] James W Demmel. *Applied numerical linear algebra.* SIAM, 1997.

[30] James W Demmel, Osni A Marques, Beresford N Parlett, and Christof Vömel. Performance and accuracy of LAPACK's symmetric tridiagonal eigensolvers. *SIAM Journal on Scientific Computing*, 30(3):1508–1526, 2008.

[31] Simplice Donfack, Jack Dongarra, Mathieu Faverge, Mark Gates, Jakub Kurzak, Piotr Luszczek, and Ichitaro Yamazaki. A survey of recent developments in parallel implementations of Gaussian elimination. *Concurrency and Computation: Practice and Experience*, 27(5):1292–1309, 2015.

[32] Jack Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. Exploiting fine-grain parallelism in recursive LU factorization. In *PARCO*, pages 429–436, 2011.

[33] Jack Dongarra, Mathieu Faverge, Thomas Herault, Mathias Jacquelin, Julien Langou, and Yves Robert. Hierarchical QR factorization algorithms for multi-core clusters. *Parallel Computing*, 39(4):212–232, 2013.

[34] Jack Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting. *Concurrency and Computation: Practice and Experience*, 26(7):1408–1431, 2014.

[35] Jack J Dongarra. Improving the accuracy of computed singular values. *SIAM Journal on Scientific and Statistical Computing*, 4(4):712–719, 1983.

[36] Jack J Dongarra, Cleve B Moler, and James Hardy Wilkinson. Improving the accuracy of computed eigenvalues and eigenvectors. *SIAM Journal on Numerical Analysis*, 20(1): 23–45, 1983.

[37] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[38] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[39] Oak Ridge Leadership Computing Facility. SUMMIT: Scale new heights. Discover new solutions. Technical report, Oak Ridge National Laboratory, 2014. URL https://www.olcf.ornl.gov/wp-content/uploads/2014/11/Summit_FactSheet.pdf.

[40] Denis Foley and John Danskin. Ultra-performance Pascal GPU and NVLink interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[41] Mark Gates, Azzam Haidar, and Jack Dongarra. Accelerating computation of eigenvectors in the dense nonsymmetric eigenvalue problem. In *International Conference on High Performance Computing for Computational Science*, pages 182–191. Springer, 2014.

[42] Gene Golub and William Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224, 1965.

[43] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[44] Ming Gu and Stanley C Eisenstat. A divide-and-conquer algorithm for the bidiagonal SVD. *SIAM Journal on Matrix Analysis and Applications*, 16(1):79–92, 1995.

[45] Azzam Haidar, Hatem Ltaief, and Jack Dongarra. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–11. IEEE, 2011.

[46] Azzam Haidar, Hatem Ltaief, Piotr Luszczek, and Jack Dongarra. A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 25–35. IEEE, 2012.

[47] Azzam Haidar, Jakub Kurzak, and Piotr Luszczek. An improved parallel singular value algorithm and its implementation for multicore hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 90. ACM, 2013.

[48] Azzam Haidar, Raffaele Solcà, Mark Gates, Stanimire Tomov, Thomas Schulthess, and Jack Dongarra. Leading edge hybrid multi-GPU algorithms for generalized eigenproblems in electronic structure calculations. In *International Supercomputing Conference*, pages 67–80. Springer, 2013.

[49] Azzam Haidar, Piotr Luszczek, and Jack Dongarra. New algorithm for computing eigenvectors of the symmetric eigenvalue problem. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1150–1159. IEEE, 2014.

[50] Azzam Haidar, Stanimire Tomov, Jack Dongarra, Raffaele Solca, and Thomas Schulthess. A novel hybrid CPU–GPU generalized eigensolver for electronic structure calculations based on fine-grained memory aware tasks. *The International Journal of High Performance Computing Applications*, 28(2):196–209, 2014.

[51] Nicholas J Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002.

[52] Jared Hoberock. Working draft, technical specification for C++ extensions for parallelism, 2014. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4071.htm.

[53] Texas Instruments. OpenMP Accelerator Model User's Guide, 2016. URL http://processors.wiki.ti.com/index.php/OpenMP_Accelerator_Model_User's_Guide.

[54] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[55] Elizabeth R Jessup and Danny C Sorensen. A divide and conquer algorithm for computing the singular value decomposition. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 61–66. Society for Industrial and Applied Mathematics, 1987.

[56] Jakub Kurzak and Jack Dongarra. Implementation of mixed precision in solving systems of linear equations on the CELL processor. *Concurrency and Computation: Practice and Experience*, 19(10):1371–1385, 2007.

[57] Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. Solving systems of linear equations on the CELL processor using Cholesky factorization. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1175–1186, 2008.

[58] Bruno Lang. Efficient eigenvalue and singular value computations on shared memory machines. *Parallel Computing*, 25(7):845–860, 1999.

[59] Hatem Ltaief, Piotr Luszczek, and Jack Dongarra. Enhancing parallelism of tile bidiagonal transformation on multicore architectures using tree reduction. *Parallel Processing and Applied Mathematics*, pages 661–670, 2012.

[60] Cleve B Moler. Iterative refinement in floating point. *Journal of the ACM (JACM)*, 14(2): 316–321, 1967.

[61] US Department of Energy. Fact Sheet: Collaboration of Oak Ridge, Argonne, and Livermore (CORAL), 2014. URL https://energy.gov/downloads/fact-sheet-collaboration-oak-ridge-argonne-and-livermore-coral.

[62] US Department of Energy. FY 2018 Department of Energy's Budget Request to Congress, 2017. URL https://energy.gov/cfo/downloads/fy-2018-budget-justification.

[63] Office of Science. U.S. Department of Energy awards $200 million for next-generation supercomputer at its Argonne National Laboratory. Technical report, US Department of Energy, 2015. URL https://energy.gov/articles/us-department-energy-awards-200-million-next-generation-supercomputer-argonne-national.

[64] OpenACC Corporation. The OpenACC™ application programming interface version 1.0, November 2011.

[65] OpenACC Corporation. Proposed additions for OpenACC 2.0, OpenACC™ application programming interface, November 2012.

[66] OpenMP Architecture Review Board. OpenMP application program interface, July 2013. Version 4.0.

[67] OpenMP Architecture Review Board. OpenMP application program interface, November 2015. Version 4.5.

[68] Scott Parker, Vitali Morozov, Sudheer Chunduri, Kevin Harms, Chris Knight, and Kalyan Kumaran. Early Evaluation of the Cray XC40 Xeon Phi System Theta at Argonne. *Cray User Group 2017 proceedings*, 2017.

[69] Michael K. Patterson. A CORAL system and implications for future hardware and data centers. Technical report, Intel, Technical Computing Systems Architecture and Pathfinding, 2014. URL https://www2.cisl.ucar.edu/sites/default/files/Patterson.pdf.

[70] John Reid. The new features of Fortran 2008. In *ACM SIGPLAN Fortran Forum*, volume 27, pages 8–21. ACM, 2008.

[71] Miroslav Rozložník, Gil Shklarski, and Sivan Toledo. Partitioned triangular tridiagonalization. *ACM Transactions on Mathematical Software (TOMS)*, 37(4):38, 2011.

[72] Satish Kumar Sadasivam, Brian W Thompto, Ron Kalla, and William J Starke. IBM Power9 processor architecture. *IEEE Micro*, 37(2):40–51, 2017.

[73] Advanced Simulation and Computing. CORAL/Sierra. Technical report, Livermore National Laboratory, 2015. URL https://asc.llnl.gov/coral-info.

[74] Balaram Sinharoy, JA Van Norstrand, Richard J Eickemeyer, Hung Q Le, Jens Leenstra, Dung Q Nguyen, B Konigsburg, K Ward, MD Brown, José E Moreira, et al. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1):2–1, 2015.

[75] Raffaele Solcà, Anton Kozhevnikov, Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Thomas C Schulthess. Efficient implementation of quantum materials simulations on distributed CPU-GPU systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 10. ACM, 2015.

[76] Nigel Stephens. ARMv8-A next-generation vector architecture for HPC. In *Hot Chips 28 Symposium (HCS), 2016 IEEE*, pages 1–31. IEEE, 2016.

[77] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. The ARM scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017.

[78] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, 1973.

[79] Brian Thompto. POWER9 processor for the cognitive era. In *Hot Chips 28 Symposium (HCS), 2016 IEEE*, pages 1–19. IEEE, 2016.

[80] Stanimire Tomov, Rajib Nath, and Jack Dongarra. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Computing*, 36(12):645–654, 2010.

[81] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[82] Wikibooks. OpenMP/Tasks - Wikibooks, open books for an open world, 2016. URL https://en.wikibooks.org/wiki/OpenMP/Tasks.

[83] Wikipedia. Mandelbrot set - Wikipedia, The Free Encyclopedia, 2017. URL https://en.wikipedia.org/wiki/Mandelbrot_set.

[84] James Hardy Wilkinson. *Rounding errors in algebraic processes*. Courier Corporation, 1994.

[85] Wei Wu, Aurelien Bouteiller, George Bosilca, Mathieu Faverge, and Jack Dongarra. Hierarchical DAG scheduling for hybrid distributed systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 156–165. IEEE, 2015.

[86] Ichitaro Yamazaki, Stanimire Tomov, Tingxing Dong, and Jack Dongarra. *High Performance Computing for Computational Science – VECPAR 2014: 11th International Conference, Eugene, OR, USA, June 30 – July 3, 2014, Revised Selected Papers*, chapter Mixed-Precision Orthogonalization Scheme and Adaptive Step Size for Improving the Stability and Performance of CA-GMRES on GPUs, pages 17–30. Springer International Publishing, Cham, 2015.

[87] Ichitaro Yamazaki, Stanimire Tomov, and Jack Dongarra. Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multiple GPUs. *SIAM Journal on Scientific Computing*, 37(3):C307–C330, 2015.