LAPACK Working Note 111, UTK, http://www.netlib.org/lapack/lawns

Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology

Jeff Bilmes[*], Krste Asanović[†], Jim Demmel[‡], Dominic Lam[§], Chee-Whye Chin [¶]

August 8, 1996

Abstract

BLAS3 operations have great potential for aggressive optimization. Unfortunately, they usually need to be hand-coded for a specific machine and compiler to achieve near-peak performance. We have developed a methodology whereby near-peak performance on a wide range of systems can be achieved automatically for such routines. First, by analyzing current machines and C compilers, we've developed guidelines for writing Portable, High-Performance, ANSI C (PHiPAC, pronounced "fee-pack"). Second, rather than code by hand, we produce parameterized code generators. Third, we write search scripts that find the best parameters for a given system. We report on a BLAS GEMM compatible multi-level cache-blocked matrix multiply generator that produces code achieving performance in excess of 90% of peak on the Sparcstation-20/61, IBM RS/6000-590, HP 712/80i, and 80% of peak on the SGI Indigo R4k. On the IBM, HP, and SGI, the resulting routine is often faster than the vendor-supplied BLAS GEMM.

1 Introduction

The use of a standard matrix-vector library interface, such as BLAS [LHKK79, DCHH88, DCDH90], enables portable application code to obtain high-performance provided that an optimized library (e.g., [AGZ94, KHM94]) is available and affordable. Developing an optimized library, however, is a difficult and time-consuming task.

Matrix-vector library routines have a large design space. Even excluding algorithmic variants such as Strassen's method [BLS91] for matrix multiplication, blocking sizes, loop nesting permutations, register allocation, and instruction scheduling, can all be varied.

The routines could be manually written in assembly code, but fully exploring the design space might be infeasible, and the resulting code might be unusable or sub-optimal on a different system.

Alternatively, the routines could be written in a high-level language and fed to an optimizing compiler. There is a large literature on relevant compiler techniques, many of which use

[*]CS Division, University of California at Berkeley. The author acknowledges the support of JSEP contract F49620-94-C-0038 and the International Computer Science Institute

[†]CS Division, University of California at Berkeley. The author acknowledges the support of ONR URI Grant N00014-92-J-1617 and the International Computer Science Institute.

[‡]CS Division and Mathematics Dept., University of California at Berkeley. The author acknowledges the support of ARPA contract DAAL03-91-C-0047 (University of Tennessee Subcontract ORA4466.02), ARPA contract DAAH04-95-1-0077 (University of Tennessee Subcontract ORA7453.02), DOE grant DE-FG03-94ER25219, DOE contract W-31-109-Eng-38, NSF grant ASC-9313958, and DOE grant DE-FG03-94ER25206.

[§]CS Division, University of California at Berkeley. The author acknowledges the support of ARPA contract DAAL03-91-C-0047 (University of Tennessee Subcontract ORA4466.02).

[¶]CS Division, University of California at Berkeley. The author acknowledges the support of ARPA contract DAAL03-91-C-0047 (University of Tennessee Subcontract ORA4466.02).

matrix-multiplication as a test case — a short list includes [WL91, LRW91, MS95, ACF95, CFH95, SMP+96].[1] While these compiler heuristics generate reasonably good code in general, they tend not to generate near-peak code for any one operation. A high-level language's semantics might also obstruct aggressive compiler optimizations. Moreover, it takes significant time and investment before compiler research appears in production compilers, so these capabilities are often simply unavailable.

We have developed a methodology, named PHiPAC, for developing Portable High-Performance matrix-vector libraries in ANSI C. Our goal is to produce, with minimal effort, high-performance matrix-vector libraries for a wide range of systems. The PHiPAC methodology has three components. First, we have developed a generic model of current C compilers and microprocessors that provides guidelines for producing portable high-performance ANSI C code. Second, rather than hand-code particular routines, we write parameterized generators [ACF95, MS95] that produce code according to our guidelines. Third, we write scripts that automatically tune code for a particular system by varying the generators' parameters and benchmarking the resulting routines.

Using this methodology, we have produced a portable, BLAS-compatible matrix multiply generator. The resulting code can achieve over 90% of peak performance on a variety of current workstations, and is often faster than the vendor-supplied optimized libraries. We concentrate on matrix multiplication in this paper, but we have produced other generators including dot-product, AXPY, and convolution, which have similarly demonstrated portable high performance.

Section 2 describes our generic C compiler and microprocessor model, and develops the resulting guidelines for writing portable high-performance C code. Section 3 describes our generator and the resulting code for a BLAS-compatible matrix multiply. Section 4 describes two strategies for searching the matrix multiply parameter space, a brute force and a more intelligent search. Section 5 lists performance results on several architectures, and, in some cases, compares them with a vendor-supplied BLAS GEMM. Section 6 lists additional generators, describes the availability of the distribution, and discusses future work.

2 PHiPAC Coding Methodology

To produce portable high-performance code, we must avoid targeting any single system. The PHiPAC machine model is therefore an abstraction of modern microprocessor-based systems together with their ANSI C compilers. The compiler must also be part of our model because we produce C code. Although ANSI C is our target language, most of the ideas behind our machine model also apply to other high level languages. By analyzing a range of machines such as workstations and microprocessor-based SMP and MPP nodes, we distilled a list of common microarchitectural features.

- Multiple integer registers, typically 32.
- Multiple floating-point registers, typically 16–32.
- A load/store architecture where it is advantageous to reuse register operands.
- A cached memory hierarchy, with up to three levels of data cache and where cache lines hold multiple words. Cache misses are costly.
- A TLB holding a limited number of page table entries, typically 64 pages of 4 KB each. TLB misses are costly.

[1] A longer list appears in [Wol96].

- The cheapest memory addressing mode is base register plus immediate offset.
- Branches are costly. Branches are cheaper using equality or zero comparisons rather than magnitude comparisons.
- Integer multiplication and division is slower than integer addition.
- Independent floating-point add, floating-point multiply, and load/store units.
- Floating-point division is slower than floating-point addition and multiplication.
- Multiple instruction latency on floating-point operations, either due to long pipelines or superscalar instruction issue.

In addition, we studied production ANSI C compilers and determined that we could usually rely on reasonable register allocation, instruction selection, and instruction scheduling. However, more sophisticated optimizations, including pointer alias disambiguation, register and cache blocking, loop unrolling, and software pipelining, were found to be best performed manually.

We emphasize that for both microarchitectures and compilers we are determining a *lowest common denominator*. Some microarchitectures or compilers will have superior characteristics in certain attributes, but, if we code assuming these exist, performance will suffer on systems where they do not. Conversely, coding for the lowest common denominator should not adversely affect performance on more capable platforms. For example, both the superscalar UltraSPARC and the single-issue MicroSPARC-II microprocessors have higher throughput when floating-point additions are interleaved with floating-point multiplications, but the SuperSPARC microprocessor can execute any permutation of independent floating-point multiplications and additions at the same rate. Also, while a few production compilers might have sophisticated loop unrolling algorithms, many do not, but a manually unrolled loop should still produce good code on a superior compiler.

2.1 PHiPAC Coding Guidlines

The following paragraphs exemplify PHiPAC coding guidelines. They can be used independently of the rest of the methodology to portably tune C application code.

Use local variables to explicitly remove false dependencies.

Casually written C code often over-specifies operation order. C compilers, constrained by C semantics, must obey these over-specifications thereby reducing optimization potential. We therefore remove these extraneous dependencies.

For example, the following code fragment contains a false Read-After-Write hazard:

```
a[i] = b[i]+c;
a[i+1] = b[i+1]*d;
```

The compiler may not assume `&a[i] != &b[i+1]` and is forced to first store `a[i]` to memory before loading `b[i+1]`. We may re-write this with explicit loads to local variables:

```
float f1,f2;
f1 = b[i]; f2 = b[i+1];
a[i] = f1 + c; a[i+1] = f2*d;
```

The compiler can now interleave execution of both original statements thereby increasing parallelism.

Exploit multiple integer and floating-point registers.

We explicitly keep values in local variables to reduce memory bandwidth demands. For example, consider the following 3-point FIR filter code:

```
while (...) {
   *res++ = filter[0]*signal[0] + filter[1]*signal[1] + filter[2]*signal[2];
   signal++;  }
```

The compiler will usually reload the filter values every loop iteration because of potential aliasing with `res`. We can remove the alias by preloading the filter into local variables that can be mapped into registers:

```
float f0,f1,f2;
f0 = filter[0]; f1 = filter[1]; f2 = filter[2];
while ( ... ) {
   *res++ = f0*signal[0] + f1*signal[1] + f2*signal[2];
   signal++;  }
```

Minimize pointer updates by striding with constant offsets.

We replace pointer updates for strided memory addressing with constant array offsets. For example:

```
f0 = *r8; r8 += 4; f1 = *r8; r8 += 4; f2 = *r8; r8 += 4;
```

should be converted to:

```
f0 = r8[0]; f1 = r8[4]; f2 = r8[8]; r8 += 12;
```

Compilers can fold the constant index into a register plus offset addressing mode.

Hide multiple instruction FPU latency with independent operations.

We use local variables to expose independent operations so they can be executed independently in a pipelined or superscalar processor. For example:

```
f1 = f5 * f9;  f2 = f6 + f10;  f3 = f7 * f11;  f4 = f8 + f12;
```

Balance the instruction mix.

A balanced instruction mix has a floating-point multiply, a floating-point add, and 1–2 floating-point loads or stores interleaved. It is not worth decreasing the number of multiplies at the expense of additions if the total floating-point operation count increases.

Aim for unit stride to exploit multi-word cache lines.

Cached machines perform poorly when accessing large data sets with non-unit stride. Whenever possible, we arrange our code to have predominantly unit-stride memory accesses. See Section 3.1, for our blocked matrix multiply example.

Convert integer multiplies to adds.

Integer multiplies and divides are slow relative to integer addition. Therefore, we use pointer updates rather than subscript expressions. Rather than:

```
for (i= ... ) { row_ptr = &p[i*col_stride]; ... }
```

we produce:

```
for (i= ... ) { ... row_ptr += col_stride; }
```

Minimize branches, avoid magnitude compares.

Branches are costly, especially on modern superscalar processors. Therefore, we unroll loops to amortize branch cost.

Also, on many microarchitectures, it is cheaper to perform equality or inequality loop termination tests than magnitude comparisons. For example, instead of:

```
for (i=0,a=start_ptr; i < ARRAY_SIZE; i++,a++} { ... }
```

we produce:

```
end_ptr = &a[ARRAY_SIZE];
for (a = start_ptr; a != end_ptr; a++) { ... }
```

This also removes one loop control variable.

Loop unroll manually to expose optimization opportunities.

We unroll loops manually to increase opportunities for other performance optimizations. For example, our 3-point FIR filter example above may be further optimized as follows:

```
float f0,f1,f2,s0,s1,s2;
f0 = filter[0]; f1 = filter[1]; f2 = filter[2];
s0 = signal[0]; s1 = signal[1]; s2 = signal[2];
*res++ = f0*s0 + f1*s1 + f2*s2;
while ( ... ) {
   signal += 3;
   s0 = signal[0]; res[0] = f0*s1 + f1*s2 + f2*s0;
   s1 = signal[1]; res[1] = f0*s2 + f1*s0 + f2*s1;
   s2 = signal[2]; res[2] = f0*s0 + f1*s1 + f2*s2;
   res += 3;
}
```

In the inner loop, there are now only two memory access per five floating point operations whereas in our unoptimized code, there were seven memory accesses per five floating point operations.

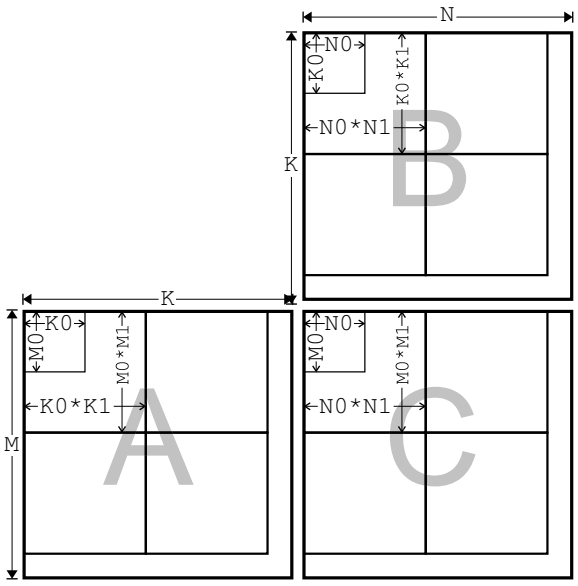

Figure 1: Matrix blocking parameters

3 Matrix Multiply Generator

`mm_gen` is a generator that produces matrix multiply code following the PHiPAC coding guidelines. It generates code for the operation $C = \alpha op(A)op(B) + \beta C$ where $op(A)$, $op(B)$, and C, are respectively M×K, K×N, and M×N matrices, α and β are scalar parameters, and $op(X)$ is either $transpose(X)$ or just X. A collection of routines produced by `mm_gen` can be used for a complete BLAS-compatible GEMM.

`mm_gen` produces a cache-blocked matrix multiply [GL89, LRW91, MS95], restructuring the algorithm for unit stride, and reducing the number of cache misses and unnecessary loads and stores. Under control of command line parameters, `mm_gen` can produce blocking code for any number of levels of memory hierarchy, including register, L1 cache, TLB, L2 cache, and so on. `mm_gen`'s code can also perform copy optimization [LRW91], optionally with a different accumulator precision.

A typical invocation of `mm_gen` is:

```
mm_gen -cb M0 K0 N0 [ -cb M1 K1 N1 ] ...
```

where the register blocking is M_0, K_0, N_0, the L1-cache blocking is M_1, K_1, N_1, etc. The parameters M_0, K_0, and N_0 are specified in units of matrix elements, i.e., single, double, or extended precision floating-point numbers, M_1, K_1, N_1 are specified in units of register blocks, M_2, K_2, and K_2 are in units of L1 cache blocks, and so on. For a particular cache level, say i, the code accumulates into a C destination block of size $M_i \times N_i$ units and uses A source blocks of size $M_i \times K_i$ units and B source blocks of size $K_i \times N_i$ units (see Figure 1). The register block parameters M_0, K_0, and N_0 determine the extent of inner loop unrolling which significantly increases optimization potential but can also significantly increase code size.

3.1 PHiPAC Matrix-Matrix Multiply Code

In this section, we examine the code produced by `mm_gen` for the operation `C = C + A*B` where A (respectively B, C) is an M×K (respectively K×N, M×N) matrix. We explain by example the code

for L1 cache blocking and register blocking where $M_0 = 3$, $K_0 = 2$, and $N_0 = 3$, and describe the scheme used for further blocking levels.

Figure 2 lists the L1 cache blocking core code comprising the 3 nested loops, M, N, and K.[2] `mm_gen` does not vary the loop permutation [MS95, LRW91] because the resulting gains in locality are subsumed by the method described below.

The outer M loop in Figure 2 maintains pointers `c0` and `a0` to rows of register blocks in the A and C matrices. It also maintains end pointers (`ap0_endp` and `cp0_endp`) used for loop termination. The middle N loop maintains a pointer `b0` to columns of register blocks in the B matrix, and maintains a pointer `cp0` to the current C destination register block. The N loop also maintains separate pointers (`ap0_0` through `ap0_2`) to successive rows of the current A source block. It also initializes a pointer `bp0` to the current B source block. We assume local variables can be held in registers, so our code uses many local pointers to minimize both memory references and integer multiplies (see Figure 3).

The K loop iterates over source matrix blocks and accumulates into the same $M_0 \times N_0$ destination block. We assume that the floating-point registers can hold a $M_0 \times N_0$ accumulator block, so this block is loaded once before the K loop begins and stored after it ends. The K loop updates the set of pointers to the A source block, one of which is used for loop termination.

Figure 4 lists the code for our fully-unrolled $3 \times 2 \times 3$ core matrix multiply. The code is not unlike the register-tiled code in [CFH95]. Local variables `c00` through `c22` hold a complete C destination block. Variables `A0` through `A2` point to successive rows of the A source matrix block, and variable `B` points to the first row of the B source matrix block. Elements in A and B are accessed using constant offsets from the appropriate pointers. Separate local variables, `_a0` through `_a2`, are used for each row of the A block to avoid false Write-After-Read hazards. This routine also updates the current B source block pointer `bp0`.

The core code performs K_0 outer products accumulating into the C destination block. The parameter K_0 controls the extent of loop-unrolling as can be seen in Figure 4. We code the outer products by loading one row of the B block, one element of the A block, then performing N_0 multiply-accumulates. The C code uses $N_0 + M_0$ memory references per $2N_0M_0$ floating-point operations in the inner K loop, while holding $M_0N_0 + N_0 + 1$ values in local variables. While the intent is that these local variables map to registers, the compiler is free to reorder all of the independent loads and multiply-accumulates to trade increased memory references for reduced register usage. The compiler also requires additional registers to name intermediate results propagating through machine pipelines.

The code we so far have described is valid only when M, K, and N are integer multiples of M_0, K_0, and N_0 respectively. We also include code that operates on power-of-two sized fringe strips, i.e., 2^0 through $2^{\lfloor \log_2 L \rfloor}$ where L is M_0, K_0, or N_0. We can therefore manage any fringe size from 1 to L$-$1 by executing an appropriate combination of fringe code. The resulting code size growth is logarithmic in the register blocking (i.e., $O(\log(M_0)\log(K_0)\log(N_0))$) yet maintains good performance. To reduce the demands on the instruction cache, we arrange the code into several independent sections, the first handling the matrix core and the remainder handling the fringes.

The separation of the row-stride and matrix dimension parameters makes it possible to implement further levels of blocking as loops around calls to lower level routines with appropriately sized sub-matrices.

Note that our procedure interface is lower level than BLAS GEMM. Each routine supports a smaller set of operations and there is no error checking. For optimal efficiency, error checking should

[2] In our terminology, the leading dimensions LDA, LDB, and LDC are called `Astride`, `Bstride`, and `Cstride` respectively.

```
void
mul_mfmf_mf(const int M, const int K, const int N,
            const float *const A, const float *const B, float *const C,
            const int Astride, const int Bstride, const int Cstride)
{
   const float *a0,*b0; float *c0;
   const float *ap0_0,*ap0_1,*ap0_2;
   const float *bp0;    float *cp0;
   const int A_sbs_stride = Astride*3;
   const int C_sbs_stride = Cstride*3;
   const int k_marg_el = K & 1;
   const int k_norm = K - k_marg_el;
   const int m_marg_el = M % 3;
   const int m_norm = M - m_marg_el;
   const int n_marg_el = N % 3;
   const int n_norm = N - n_marg_el;
   float *const c0_endp = C+m_norm*Cstride;
   register float c00,c01,c02,c10,c11,c12,c20,c21,c22;
   for (c0=C,a0=A; c0!= c0_endp; c0+=C_sbs_stride,a0+=A_sbs_stride) {
      const float* const ap0_endp = a0 + k_norm;
      float* const cp0_endp = c0 + n_norm;
      for (b0=B,cp0=c0; cp0!=cp0_endp; b0+=3,cp0+=3) {
         ap0_0 = a0;
         ap0_1 = ap0_0 + Astride;
         ap0_2 = ap0_1 + Astride;
         bp0=b0;
         LOAD3x3(c00,c01,c02,c10,c11,c12,c20,c21,c22,cp0,Cstride);
         for (; ap0_0!=ap0_endp; ap0_0+=2,ap0_1+=2,ap0_2+=2) {
            mul_mf3x2mf2x3_mf3x3(c00,c01,c02,c10,c11,c12,c20,c21,c22,
                                 ap0_0,ap0_1,ap0_2,bp0,Bstride);
         }
         STORE3x3(c00,c01,c02,c10,c11,c12,c20,c21,c22,cp0,Cstride);
      }
   }
}
```

Figure 2: $M_0 = 3$, $K_0 = 2$, $N_0 = 3$ matrix multiply L1 routine.

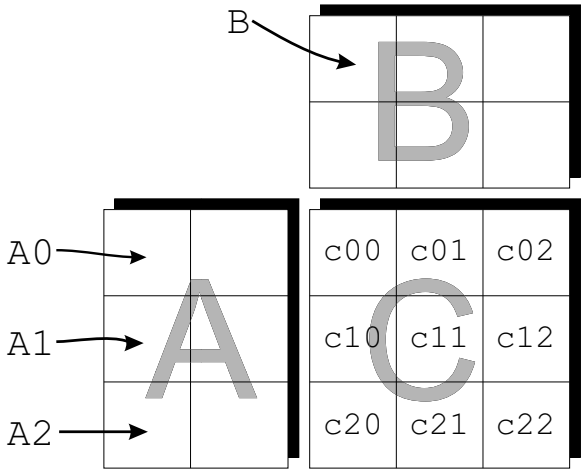

Figure 3: $M_0 = 3$, $K_0 = 2$, $N_0 = 3$ register-blocked matrix multiply diagram.

be performed by the caller when necessary rather than unnecessarily by the callee. We nevertheless have implemented a FORTRAN BLAS GEMM compatible interface to our routines [BAD+].

4 Search Scripts

For each combination of generator parameters and compilation options, the PHiPAC matrix multiply search script calls the generator, compiles the resulting routine, links it with timing code, and benchmarks the resulting executable. In our search scripts, we assume that we have all machine specific information, such as the number of integer and floating-point registers and sizes of each level of cache, available at the start of the search.

4.1 Naive Parameter Search

In the initial PHiPAC alpha release [BAD+], we use a simple brute force search. To produce a full BLAS GEMM routine, we need to search three cases of $op(A) \times op(B)$ separately: $A \times B$, $A^T \times B$, and $A \times B^T$ ($A^T \times B^T$ is the mirror image of $A \times B$). For each transposition case, we search the blocking parameters for register, L1 cache, and L2 cache.

The register block search evaluates all combinations of M_0 and N_0 where $\mathrm{NR}/4 \leq M_0 N_0 \leq \mathrm{NR}$ and where NR is the number of machine floating-point registers. We search the above for $1 \leq K_0 \leq K_0^{max}$ where $K_0^{max} = 20$ but is adjustable. Empirically, $K_0^{max} > 20$ has never been beneficial.

For the register block search, we do not want L1 cache effects to influence performance. Therefore, for each set of blocking parameters, we benchmark all square matrices $M = K = N = D$, where D runs over powers of 2, powers of 3, multiples of 10, and primes such that three D×D matrices fit in L1 cache. From these timings we pick the best register blocking.

We perform the L1 cache blocking search after the best register blocking is known. The L1 blocks should all fit in L1 cache. To reduce the number of L1 block boundaries, they should be made larger,

```
#define mul_mf3x2mf2x3_mf3x3(c00,c01,c02,c10,c11,c12,c20,c21,c22, \
                             A0,A1,A2,B,Bstride) \
{ \
   register float _b0,_b1,_b2; \
   register float _a0,_a1,_a2; \
   \
   _b0 = B[0]; _b1 = B[1]; _b2 = B[2]; \
   B += Bstride; \
   _a0 = A0[0]; \
   c00 += _a0*_b0; c01 += _a0*_b1; c02 += _a0*_b2; \
   _a1 = A1[0]; \
   c10 += _a1*_b0; c11 += _a1*_b1; c12 += _a1*_b2; \
   _a2 = A2[0]; \
   c20 += _a2*_b0; c21 += _a2*_b1; c22 += _a2*_b2; \
   \
   _b0 = B[0]; _b1 = B[1]; _b2 = B[2]; \
   B += Bstride; \
   _a0 = A0[1]; \
   c00 += _a0*_b0; c01 += _a0*_b1; c02 += _a0*_b2; \
   _a1 = A1[1]; \
   c10 += _a1*_b0; c11 += _a1*_b1; c12 += _a1*_b2; \
   _a2 = A2[1]; \
   c20 += _a2*_b0; c21 += _a2*_b1; c22 += _a2*_b2; \
}
```

Figure 4: $M_0 = 3$, $K_0 = 2$, $N_0 = 3$ register blocked matrix multiply code.

but to minimize the probability of cache misses [LRW91], they should be made smaller. For the D×D square case, this occurs roughly when $3D^2 = L1$ where $L1$ is the L1 cache size. We therefore search this neighborhood setting M_1 to the values $a \times D/M_0$ where $a \in 0.25, 0.5, 1.0, 1.5, 2.0$ and $D = \sqrt{L1/3}$ (K_1 and N_1 are set similarly resulting in 125 combinations). The matrix sizes used to test these blocking parameters either fit in L2 cache, or are within some upper bound if no L2 cache exists.

Unfortunately, this search strategy takes too long. Generated code can be lengthy and under full optimization, compilation time is significant. Timing all matrix sizes is also time-consuming, particularly where the machine timer resolution is poor. We had initially hoped search time would be unimportant because, once the parameters were known for a given system, they could be globally published on the web. In fact, new machines and compilers are being introduced fairly often, and some people are interested in finding optimal parameters for their own matrix sizes. Furthermore, parameter searching for a complete BLAS implementation should not take longer than the useful life of the architecture. The next section describes a better search strategy.

4.2 Smarter Register Block Search

The core register-blocked code is the most important to optimize because it performs the majority of the computation. The core code, however, consumes less then one fourth of the total code size. Therefore, in our newer strategy, we produce code containing only the matrix multiply core and benchmark only matrix sizes that are multiples of the register block size. This strategy produces more accurate core performance results since timing values are not distorted by fringe code.

We do not reduce the space of possible values for M_0, K_0, and N_0, but we believe performing a best-first rather than a naive search should produce better numbers sooner. That is, we search first the unseen neighbors of the best parameters seen so far, where the neighbors of a particular tuple (M_0, K_0, N_0) are defined as ($\{M_0, M_0 \pm 1\}$, $\{K_0, K_0 \pm 1\}$, $\{N_0, N_0 \pm 1\}$). We can further reduce search time by terminating early after reaching an acceptable efficiency.

While benchmarking each set of register blocking parameters, we fit a model of in-cache matrix multiply performance to the timing figures. We will later use this information to both find good L1 blocking sizes and also to find the best portable routine across a range of systems. Observe the basic structure of the core code:

```
initialization O(1)
for M
  O(M) code
  for N
    O(M*N) code
    for K
       O(M*N*K) code
```

If the three matrices fit in L1 cache, and the matrix dimensions are multiples of the register block sizes, the following simple four parameter linear model can accurately predict the running time:

$$T = a_0 + a_1 \frac{M}{M_0} + a_2 \frac{MN}{M_0 N_0} + a_3 \frac{MNK}{M_0 N_0 K_0}$$

where T is the total time to execute an M×K×N matrix multiply.

We can estimate the parameters a_0 through a_3 by timing four matrix sizes and solving the resulting system of four equations and four unknowns. We choose the following sizes to produce a well-conditioned system of equations:

	Sparcstation-20/61	SGI Indigo R4K	HP 712/80i	IBM RS/6000-590
Processor	SuperSPARC+	R4000	PA7100LC	RIOS-2
Frequency (MHz)	60	100	80	71.5
Max Instructions/cycle	3	1	2	6
Peak MFLOPS (32b/64b)	60/60	67/50	160/80	266/266
FP registers (32b/64b)	32/16	16/16	64/32	32/32
L1 D-cache (KB)	16	8	128	256
L2 D-cache (KB)	1024	1024	-	-
OS	SunOS 4.1.3	Irix 4.0.5H	HP-UX 9.05	AIX 3.2
C Compiler	Sun acc 2.0.1	SGI cc 3.10	HP c89 9.61	IBM xlc 1.3

Table 1: Workstation system details.

- $M = M0$, $N = N0$, $K = K0$ giving information about a_0 through a_3.
- $N = N0$, $K = K0$, and M set such that A and C together are roughly the size of L1 cache. We get information primarily about a_1 through a_3.
- $M = M0$, $K = K0$, and N set such that B and C together are roughly the size of L1 cache. We get information primarily about a_2 and a_3.
- $M = M_0$, $N = N_0$, and K set such that A and B together are roughly the size of L1 cache. We get information primarily about a_3.

To summarize, the advantages of the new searching strategy include smaller code that takes much less time to compile, more accurate core timing, better numbers sooner, fewer sample points used for each register blocking parameter set, and derivation of a timing model for in-cache matrix-multiply.

A preliminary implementation of the optimized search strategy shows a significant reduction in search time (from 24 hours down to 5) and produces comparable results.

5 Results

To evaluate our methodology, we chose four commercial workstation systems with different instruction set architectures and widely varying microarchitectures and memory hierarchies. Table 1 summarizes pertinent details.

For each system, we ran the PHiPAC alpha release search script to find the best register and L1 cache blocking parameters which are shown in Table 2 together with the compiler options used.

	Sparcstation-20/61	SGI Indigo R4K	HP 712/80i	IBM RS/6000-590
M_0,K_0,N_0	2,4,10	2,10,3	4,1,4	2,1,10
M_1,K_1,N_1	26,10,4	30,4,10	25,100,25	105,70,28
CFLAGS	-fast	-O2 -mips2	+O2	-O3 -qarch=pwr2

Table 2: Parameters for the best matrix multiply found by PHiPAC search.

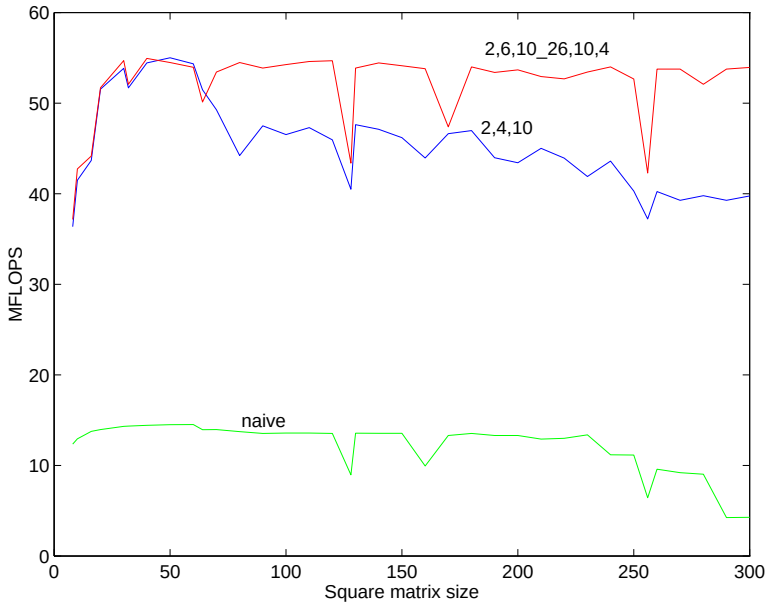

Figure 5: Performance of single precision matrix multiply on a Sparcstation-20/61. The bottom "naive" plot shows the performance for code containing three nested loops. The "2,4,10" plot shows PHiPAC performance with just register blocking. The last plot shows the performance for the PHiPAC L1 blocked routine.

Figures 5–8 plot the resulting performance for each system. The results are all for single precision, except the IBM RS/6000-590 which uses double precision. We show data for the square matrix sizes described in section 4. We include vendor-optimized BLAS GEMM performance where available.

In each case, the PHiPAC yields a substantial fraction of peak performance, and is competitive with the vendor BLAS. The PHiPAC matrix multiply routines have a simpler interface than BLAS GEMM, but the standard GEMM argument checking should not appreciably influence performance, especially on larger matrix sizes.

The PHiPAC methodology can also improve performance even if there is no scope for memory blocking. In Figures 9 and 10 we plot the performance of a dot product code generated using PHiPAC techniques. Although the parameters used were obtained using a short manual search, we can see a significant performance boost over naive code. For the SGI, we are competitive with the assembly coded vendor BLAS.

The PHiPAC routines occasionally suffer from cache conflict effects. Our measurements exaggerate this effect by including all power-of-2 sized matrices, and by allocating all regions contiguously in memory. A drawback of the PHiPAC approach is that we can not control the order compilers schedule independent loads. We've occasionally found that exchanging two loads in the assembly output can halve the number of cache misses where conflicts occur, without otherwise impacting performance.

6 Status, Availability, and Future Work

This paper has demonstrated our ability to write fast matrix-vector code in portable ANSI C via parameterized code generators. Library writers might achieve significant improvements by writing

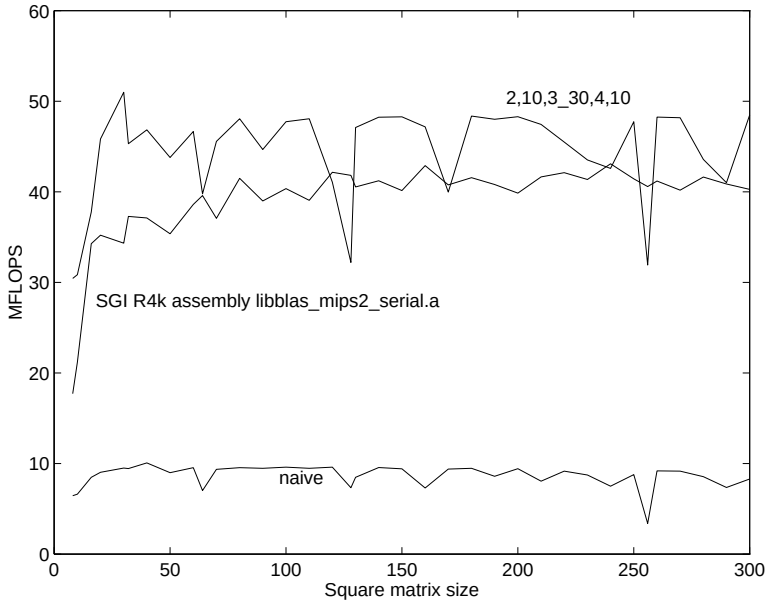

Figure 6: Performance of single precision matrix multiply on an 100 MHz SGI Indigo R4K. The middle plot is the SGI-supplied mips2 assembly-coded SGEMM available from their ftp site. The top plot is the PHiPAC generated code.

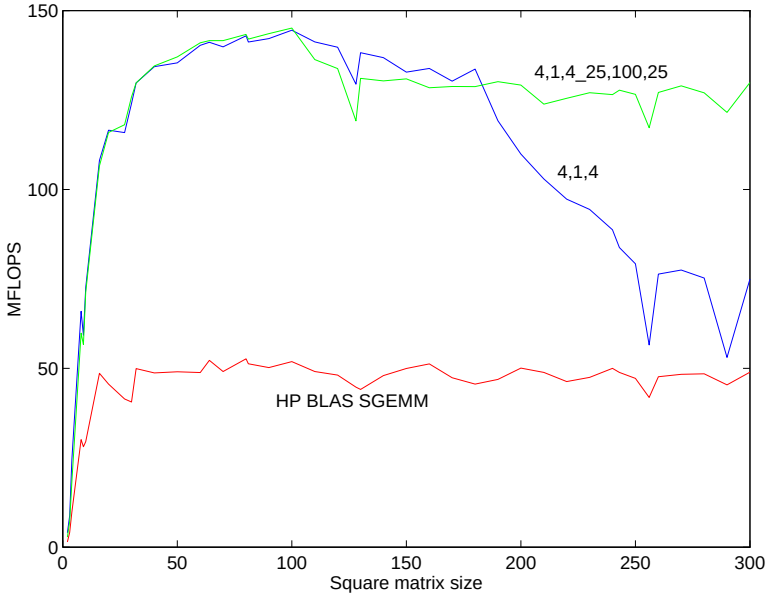

Figure 7: Performance of single precision matrix multiply on a HP 712/80i. The bottom plot is the HP-supplied SGEMM from `libvec.a` in the compiler distribution. The "4,1,4" plot is the performance of the PHiPAC routine with just register blocking. The last plot is the PHiPAC routine with both register and L1 cache blocking.

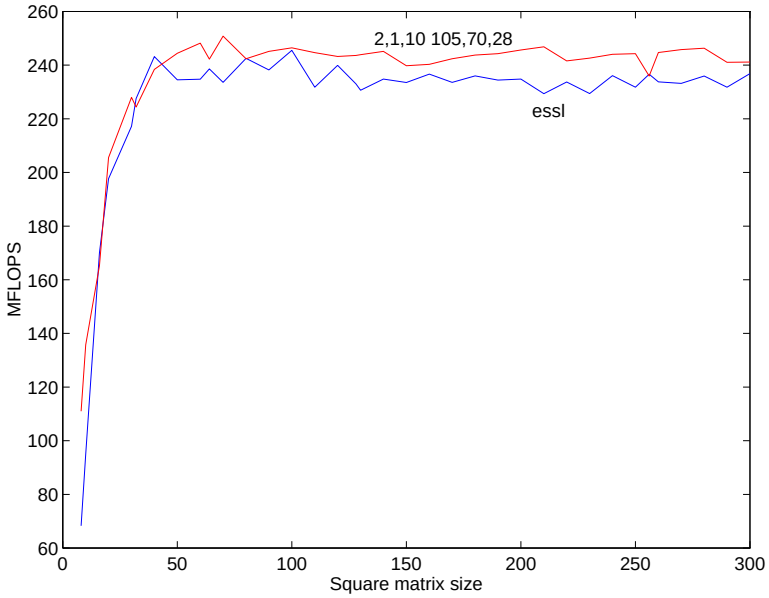

Figure 8: Performance of double precision matrix multiply on an IBM RS/6000-590. The bottom plot is IBM's ESSL DGEMM.

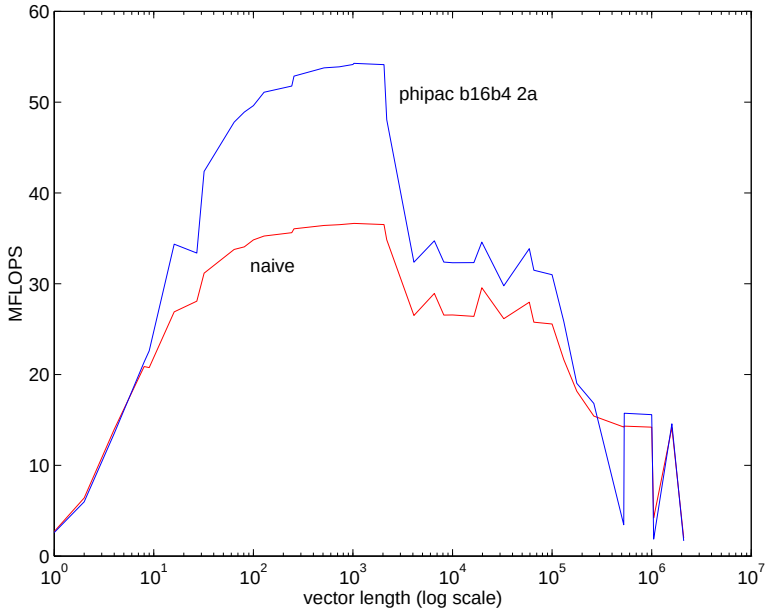

Figure 9: Performance of single precision unit-stride dot-product on a Sparcstation-20/61. The bottom curve is the performance of a simple loop. The top curve is the performance of PHiPAC generated code.

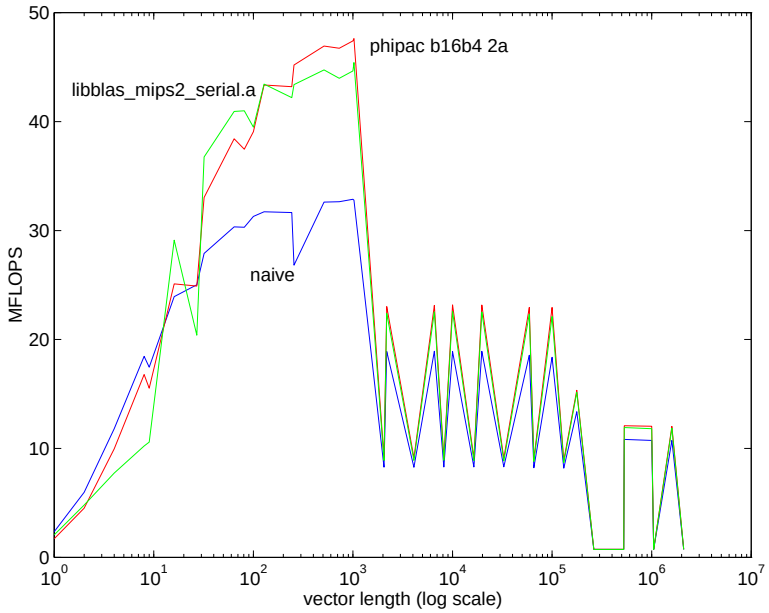

Figure 10: Performance of single precision unit-stride dot-product on a 100 MHz SGI R4k. The bottom curve is the performance of a simple loop. The other two curves are the vendor-supplied SDOT routine, and the PHiPAC generated code.

a generator and searching a parameter space, instead of coding directly in a target language.

The PHiPAC alpha release [BAD+] contains the matrix multiply generator, the naive search scripts written in `perl`, and timing libraries. We have also written parameterized generators for matrix-vector and vector-matrix multiply, dot product, AXPY, convolution, and outer-product. We have created a Web site on which we plan to list blocking parameters for many systems.

In future work, we will release the smarter search scripts. We plan to use the a_i coefficients described in Section 4.2 to determine good L1 blocking sizes. We also plan to use the same coefficients to find a parameter set that works well on a set of systems. At some point, PHiPAC will be integrated with Bo Kågström's GEMM based BLAS3 package [BLL93]. The PHiPAC routines might also be integrated into LAPACK [ABB+92].

In the more distant future, the matrix multiply code will dynamically adjust its L1 blocking size according to various criteria [LRW91]. This will increase performance for certain pathological combinations of matrix dimensions and cache structure. Other generators, such as FFT and sort, are planned.

We wish to thank the International Computer Science Institute for its support. We also wish to thank Nelson Morgan who provided the initial impetus for this project.

References

[ABB+92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. LAPACK users' guide, release 1.0. In *SIAM*, Philadelphia, 1992.

[ACF95] B. Alpern, L. Carter, and J. Ferrante. Space-limited procedures: A methodology for portable high-performance. In *International Working Conference on Massively Parallel*

Programming Models, 1995.

[AGZ94] R.C. Agarwal, F.G. Gustavson, and M. Zubair. *IBM Engineering and Scientific Subroutine Library, Guide and Reference*, 1994. Available through IBM branch offices.

[BAD+] J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C.W. Chin. The PHiPAC WWW home page. `http://www.icsi.berkeley.edu/~bilmes/phipac`.

[BLL93] B.Kågström, P. Ling, and C. Van Loan. Portable high performance GEMM-based level 3 blas. In R.F. Sincovec et al., editor, *Parallel Processing for Scientific Computing*, pages 339–346, Philadelphia, 1993. SIAM Publications.

[BLS91] D. H. Bailey, K. Lee, and H. D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *J. Supercomputing*, 4:97–371, 1991.

[CFH95] L. Carter, J. Ferrante, and S. Flynn Hummel. Hierarchical tiling for improved superscalar performance. In *International Parallel Processing Symposium*, April 1995.

[DCDH90] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[DCHH88] J. Dongarra, J. Du Cros, S. Hammarling, and R.J. Hanson. An extended set of FORTRAN basic linear algebra subroutines. *ACM Trans. Math. Soft.*, 14:1–17, March 1988.

[GL89] G.H. Golub and C.F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 1989.

[KHM94] C. Kamath, R. Ho, and D.P. Manley. DXML: A high-performance scientific subroutine library. *Digital Technical Journal*, 6(3):44–56, Summer 1994.

[LHKK79] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[LRW91] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[MS95] J.D. McCalpin and M. Smotherman. Automatic benchmark generation for cache optimization of matrix algorithms. In R. Geist and S. Junkins, editors, *Proceedings of the 33rd Annual Southeast Conference*, pages 195–204, New York, NY, March 1995. Association for Computing Machinery, ACM.

[SMP+96] R. Saavedra, W. Mao, D. Park, J. Chame, and S. Moon. The combined effectiveness of unimodular transformations, tiling, and software prefetching. In *Proceedings of the 10th International Parallel Processing Symposium.* IEEE Computer Society, April 15–19 1996.

[WL91] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[Wol96] M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley, 1996.