# PULSAR Users' Guide

## Parallel Ultra-Light Systolic Array Runtime

Version 2.0

November, 2014

LAPACK Working Note 288

Technical Report UT-EECS-14-733

Electrical Engineering and Computer Science
**University of Tennessee**

Jack Dongarra
Jakub Kurzak
Piotr Luszczek
Ichitaro Yamazaki

*And now I see with eye serene*
*The very pulse of the machine.*
*– William Wordsworth*

# Contents

# Preface

PULSAR version 2.0, released in November 2014, is a complete programming platform for large-scale distributed memory systems with multicore processors and hardware accelerators. PULSAR provides a simple abstraction layer over multithreading, message passing, and multi-GPU, multi-stream programming. PULSAR offers a general-purpose programming model, suitable for a wide range of scientific and engineering applications. PULSAR was inspired by systolic arrays, popularized by Hsiang-Tsung Kung and Charles E. Leiserson [1, 2, 3].

# CHAPTER 1

# Essentials

PULSAR is a programming model and a runtime scheduling system implementing that model, that are inspired by systolic arrays. The name PULSAR is an acronym for *Parallel Ultra Light Systolic Array Runtime*. PULSAR project website is located at:

http://icl.utk.edu/pulsar/

## 1.1 Problems for which PULSAR is Suitable

PULSAR is suitable for implementing any algorithm, for which the message-passing paradigm is suitable. In the past, PULSAR has been used mainly to implement dense linear algebra algorithms. Currently, the PULSAR team targets deep learning algorithms. PULSAR is also a natural match for stencil computations.

## 1.2 Computers for which PULSAR is Suitable

PULSAR is suitable for any parallel computer, as it offers efficient multithreading for multicore processors, multi-GPU, multi-stream execution on hybrid computers, and message-passing on distributed-memory systems.

## 1.3 Software Components that PULSAR Requires

PULSAR requires POSIX threads for multithreading, NVIDIA CUDA for multi-GPU programming and MPI for message-passing. However, PULSAR can be built without CUDA and without MPI for testing and prototyping.

## 1.4 Availability of PULSAR

PULSAR is distributed as source code and is meant to be compiled from source on the host system. In certain cases, a pre-built binary may be provided along with the source code. Such packages, built by the PULSAR developers, will be provided as separate archives on the PULSAR download page:

http://icl.utk.edu/pulsar/software/

The PULSAR team does not reserve an exclusive right to provide such packages. They can be provided by other individuals or institutions. However, in case of problems with binary distributions acquired from other places, the provider needs to be asked for support rather than the PULSAR developers.

## 1.5 Documentation of PULSAR

The PULSAR package comes with a variety of documentation including:

- The PULSAR Users' Guide (this document)
- The PULSAR Reference Manual
- The PULSAR README
- The PULSAR Release Notes
- The PULSAR Installation Instructions

You will find all of these in the documentation section on the PULSAR website:

http://icl.utk.edu/pulsar/ → Documentation

## 1.6 Commercial Use of PULSAR

PULSAR is a freely available software package with a license that allows its use or inclusion in commercial packages. The PULSAR team asks only that proper credit be given by citing this users' guide as the official reference for PULSAR.

Like all software, this package is copyrighted. It is not trademarked. However, if modifications are made that affect the interface, functionality, or accuracy of the resulting software, the name of the routine should be changed and the modifications to the software should be noted in the modifier's documentation.

The PULSAR team will gladly answer questions regarding this software. If modifications are made to the software, however, it is the responsibility of the individual or institution who modified the routine to provide the support.

## 1.7 PULSAR Support

PULSAR has been thoroughly tested before its release by using multiple combinations of machine architectures, compilers and libraries. The PULSAR project supports the package in the sense that reports of errors or poor performance will gain immediate attention from the developers. Such reports – and also descriptions of interesting applications and other comments – should be posted to the PULSAR's User Forum:

http://icl.utk.edu/pulsar/forum/

## 1.8 PULSAR Funding

PULSAR was funded by the National Science Foundation, under grant #1117062 entitled *SHF: Small: Parallel Unified Linear algebra with Systolic ARrays (PULSAR)*.

## 1.9 Hardware Allocations for PULSAR

Computing cycles for the development of PULSAR have been mainly provided by the National Institute for Computational Science (NICS).

# CHAPTER 2

# Fundamentals

The PULSAR programming model relies on the following five abstractions to define the processing pattern:

**Virtual Systolic Array (VSA)** is a set of VDPs connected with channels.

**Virtual Data Processor (VDP)** is the basic processing element in the VSA.

**Channel** is a point-to-point connection between a pair of VDPs.

**Packet** is the basic unit of information transferred in a channel.

**Tuple** is a unique VDP identifier.

It also relies on the following two abstractions to map the processing pattern to the actual hardware:

**Thread** is synonymous with a CPU thread or a collection of threads.

**Device** is synonymous with an accelerator device (GPU, Xeon Phi, etc.)

The sections to follow describe the roles of the different entities, how the VDP operation is defined, how the VSA is constructed, and how the VSA is mapped to the hardware. These operations are accessible to the user through PULSAR's *Application Programming Interface* (API), which is currently available with C bindings.

## 2.1 Tuple

Tuples are strings of integers. Each VDP is uniquely identified by a tuple. Tuples can be of any length, and different length tuples can be used in the same VSA. Two tuples are identical if they are of the same length and have identical values of all components. Tuples are created using the variadic function `prt_tuple_new()`, which takes a (variable length) list of integers as its input. The user only creates tuples. After creation, tuples are passed to VDP constructors and channel constructors. They are destroyed by the runtime at the time of destroying those objects. As a general rule in PULSAR, the user only creates objects, and looses their ownership after passing them to the runtime.

## 2.2 Packet

Packets are basic units of information exchanged through channels connecting VDPs. A packet contains a reference to a continuous piece of memory of a given size. Conceptually, packets are created by VDPs. The user can use the VDP function `prt_vdp_packet_new()` to create a new packet. A packet can be created from preallocated memory by providing the pointer. Alternatively, new memory can be allocated by providing a NULL pointer. The VDP can fetch a packet from an input channel using the function `prt_vdp_channel_pop()`, and push a packet to an output channel using the function `prt_vdp_channel_push()`. The VDP does not loose the ownership of the packet after pushing it to a channel. The packet can be used until the `prt_vdp_packet_release()` function is called, which discards it.

## 2.3 Channel

Channels are unidirectional point-to-point connections between VDPs, used to exchange packets. Each VDP has a set of input channels and a set of output channels. Packets can be fetched from input channels and pushed to output channels. Channels in each set are assigned consecutive numbers starting from zero (*slots*). Channels are created by using the `prt_channel_new()` function. The user does not destroy channels. The runtime destroys channels at the time of destroying the VDP. After creation, each channel has to be inserted in the appropriate VDP, using the `prt_vdp_channel_insert()` function. The user has to insert a full set of channels into each VDP. At the time of inserting the VDP in the VSA, the system joins channels that identify the same communication path.

## 2.4 Virtual Data Processor

The VDP is the basic processing element of the VSA (Figure 2.1). Each VDP is uniquely identified by a tuple and assigned a function which defines its operation. Within that function, the VDP has access to a set of global parameters, its private, persistent local storage, and its channels. The runtime invokes that function when there are packets in all of the VDP's input channels. This is called *firing*. When the VDP fires, it can fetch packets from its input channels, call computational kernels, and push packets to its output channels. It is not required that these operations are invoked in any particular order. The VDP fires a prescribed number of times. When the VDP's counter goes down to zero, the VDP is destroyed. The VDP has access to its tuple and its counter. Figure 2.2 shows simple VDP processing patterns.

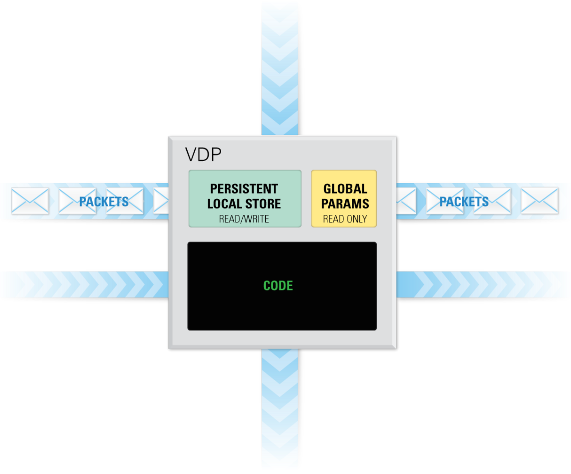

Figure 2.1: Virtual Data Processor.

At the time of the VDP creation, the user specifies if the VDP resides on a CPU or on an accelerator. This is an important distinction, because the code of a CPU VDP has synchronous semantics, while the code of an accelerator VDP has asynchronous semantics. For a CPU VDP, actions are executed as they are invoked, while for an accelerator VDP, actions are queued for execution after preceding actions complete. In the CUDA implementation, each VDP has its own stream. All kernel invocations have to be asynchronous calls, placed in the VDP's stream. The runtime will also place all channel operations in the VDP's stream.

```
    prt_packet_t *packet = prt_vdp_packet_new(vdp, ...);
    kernel_that_writes(..., packet->data, ...);
    prt_vdp_channel_push(vdp, slot, packet);
    prt_vdp_packet_release(vdp, packet);

    prt_packet_t *packet = prt_vdp_channel_pop(vdp, slot);
    kernel_that_modifies(..., packet->data, ...);
    prt_vdp_channel_push(vdp, slot, packet);
    prt_vdp_packet_release(vdp, packet);

    prt_packet_t *packet = prt_vdp_channel_pop(vdp, slot);
    prt_vdp_channel_push(vdp, slot, packet);
    kernel_that_reads(..., packet->data, ...);
    prt_vdp_packet_release(vdp, packet);
```

Figure 2.2: Simple VDP processing patterns.

## 2.5 Virtual Systolic Array

VSA contains all VDPs and their channel connections (Figure 2.3), and stores the information about the mapping of VDPs to the hardware. The VSA needs to be created first and then launched. An empty VSA is created using the `prt_vsa_new()` function. Then VDPs can be inserted in the VSA using the `prt_vsa_vdp_insert()` function. Then the VSA can be executed using the `prt_vsa_run()` function, and then destroyed using the `prt_vsa_delete()` function. Figure 2.4 shows the basic VSA construction and execution process.

At the time of creation, using the `prt_vsa_new()` function, the user provides the number of CPU threads to launch per each distributed memory node, and the number of accelerator devices to use per each node. The user also provides a function for mapping VDPs to threads and devices. The function takes as parameters: the VDP's tuple, the total number of threads, and the total number of devices, and returns a structure indicating if the VDP is assigned to a thread or a device, and the global rank of the thread or device, where the VPD resides.

VSA construction can be replicated or distributed. The replicated construction is more straightforward, from the user's perspective. In the replicated construction, each MPI process inserts all the VDPs, and the system filters out the ones that do not belong in a given node, based on the mapping function. However, the VSA construction process is inherently distributed, so each process can also insert only the VDPs that belong in that process.

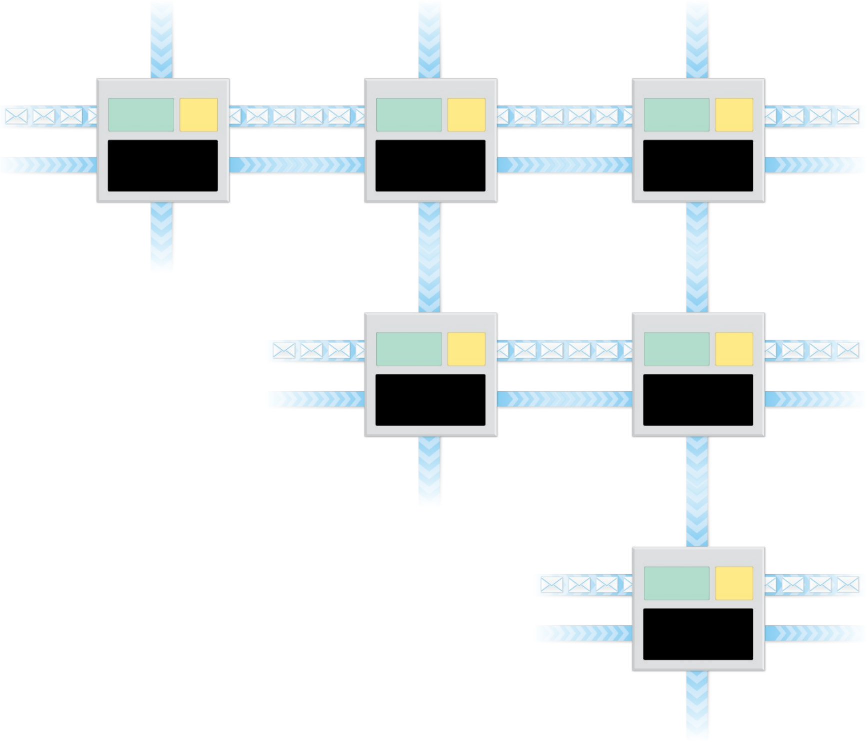

Figure 2.3: Virtual Systolic Array.

```
prt_vsa_t *vsa = prt_vsa_new(num_threads, num_devices, ...);

for (v = 0; v < vdps; v++) {
    prt_vdp_t *vdp = prt_vdp_new(...);
    for (in = 0; in < inputs; in++) {
        prt_channel_t *input = prt_channel_new(...);
        prt_vdp_channel_insert(vdp, input, ...);
    }
    for (out = 0; out < outputs; out++) {
        prt_channel_t *output = prt_channel_new(...);
        prt_vdp_channel_insert(vdp, output, ...)
    }
    prt_vsa_vdp_insert(vsa, vdp, ...);
}
double time = prt_vsa_run(vsa);
prt_vsa_delete(vsa);
```

Figure 2.4: Basic VSA construction process.

# CHAPTER 3

# Further Details

## 3.1 VSA Construction

### 3.1.1 VSA Creation, Execution and Deletion

An empty VSA is created using the prt_vsa_new() function. The function takes as parameters: the number of threads per node, the number of devices per node, a pointer to the global store, accessible to all VDPs in a read-only mode, and a pointer to the function for mapping VDPs to threads and devices. The function returns a pointer to a new VSA with an empty set of VDPs. See section 3.1.4 for the description of the parameters of the mapping function. If MPI is used, MPI_Init() has to be called before prt_vsa_new().

```
prt_vsa_t *prt_vsa_new(
    int num_threads,
    int num_devices,
    void* global_store,
    prt_mapping_t (*)(int*, void*, int, int)vdp_mapping)
```

After creation, the VSA has to be populated with VDPs, as described in section 3.1.2. Then the VSA can be launched using the prt_vsa_run() function. The function takes a pointer to the VSA as the parameter and returns the execution time in seconds as a double precision floating point number.

```
double prt_vsa_run(prt_vsa_t *vsa)
```

The VSA has a few configuration parameters, which can be changed at any time after the creation of the VSA and before the launch of the VSA, as described in section 3.1.5.

After execution, the VSA can be destroyed using the `prt_vsa_delete()` function. The function takes a pointer to the VSA as the parameter and destroys all resources associated with the VSA.

```
void prt_vsa_delete(prt_vsa_t *vsa)
```

### 3.1.2 VDP Creation and Insertion

A VDP is created using the `prt_vdp_new()` function. The function takes as parameters: the VDP's tuple, the VDP's counter (the number of times the VDP will be fired), a pointer to the function implementing the VDP's operation, the size in bytes of the VDP's local store, the number of input channels, the number of output channels, and the RGB color for tracing, when using PULSAR's internal tracing (section 3.1.5). The function returns a pointer to a new VDP.

```
prt_vdp_t *prt_vdp_new(
    int *tuple,
    int counter,
    prt_vdp_function_t vdp_function,
    size_t local_store_size,
    int num_inputs,
    int num_outputs,
    int color)
```

After creation, the VDP has to be populated with channels, as described in section 3.1.3. Then the VDP can be inserted into the VSA using the `prt_vsa_vdp_insert()` function. The function takes a pointer to the VSA and a pointer to the VDP as parameters.

```
void prt_vsa_vdp_insert(prt_vsa_t *vsa, prt_vdp_t *vdp)
```

The user does not free the VDP. At the time of calling `prt_vsa_vdp_insert()`, the runtime takes ownership of the VDP. The VDP will be destroyed in the process of the VSA's execution or at the time of calling `prt_vsa_delete()`. User's attempt to free the VDP are likely to cause a crash.

The user has to define the VDP's function. The runtime invokes that function when packets are available in all of the VDP's channels, which is called *firing*.

```
void vdp_function(prt_vdp_t *vdp)
```

Inside that function, the user has access to the VDP object. In particular, the user has access to the following fields:

```
void *global_store
void *local_store
int *tuple
int counter
prt_location_t location
cudaStream_t stream
```

`global_store` is the read-only global storage area, passed to the VSA at the time of creation (section 3.1.1). `local_store` is the VDP's private local storage areal, which is persistent between firings. `tuple` is the VDP's unique tuple, assigned at the time of creation. `counter` is the VDP's counter. At the first firing, the counter is equal to the value assigned at the time of the VDP's creation. At each firing the counter is decremented by one. At the last firing the counter is equal one. `location` indicated if the VDP is a CPU VDP or a GPU VDP. The value `PRT_LOCATION_HOST` indicates a CPU. The value `PRT_LOCATION_DEVICE` indicates a GPU. Finally, `stream` contains the CUDA stream of a GPU VDP.

### 3.1.3 Channel Creation and Insertion

A channel is created using the `prt_channel_new()` function. The function takes a parameters: the size of the channel, which indicates the maximum size of packets intended to be transmitted in that channel, the tuple of the source VDP (the sender), the slot number in the source VDP, the tuple of the destination VDP (the receiver), and the slot number in the destination VDP.

```
prt_channel_t* prt_channel_new(
    size_t size,
    int *src_tuple,
    int src_slot,
    int *dst_tuple,
    int dst_slot)
```

After creation, the channel can be inserted into the a VDP using the `prt_vdp_channel_insert()` function. The function takes as parameter: the pointer to the VDP, the pointer to the channel, the direction of the channel from the standpoint of that VDP, and the slot number in that VDP.

```
void prt_vdp_channel_insert(
    prt_vdp_t *vdp,
    prt_channel_t *channel,
    prt_channel_direction_t direction,
    int slot)
```

The user does not free the channel. At the time of calling prt_vdp_channel_insert(), the runtime takes ownership of the channel. The channel will be destroyed in the process of the VSA's execution or at the time of calling prt_vsa_delete(). User's attempt to free the channel are likely to cause a crash.

### 3.1.4 Mapping of VDPs to Threads and Devices

The user defines the placement of VDPs on CPUs and GPUs by providing the mapping function at the time of the VSA creation with prt_vsa_new() (section 3.1.1). The runtime calls that function for each VDP and passes as parameters: the VDP's tuple, the pointer to the global store, the total number of CPU threads at the VSA's disposal in that launch, and the total number of devices in that launch.

```
prt_mapping_t vdp_mapping(
    int *tuple,
    void *global_store,
    int total_threads,
    int total_devices)
```

The function has to return the mapping information in an object of type prt_mapping_t, with the fields location and rank, where the location can be either PRT_LOCATION_HOST or PRT_LOCATION_DEVICE, and the rank indicates the global rank of the unit.

```
typedef struct prt_mapping_s {
    prt_location_t location;
    int rank;
} prt_mapping_t;
```

### 3.1.5 VSA Configuration

The user can set the VSA's configuration parameters by calling the prt_vsa_config_set() function. The function takes as parameters: the pointer to the VSA, the name of the parameter, and the value to set. Currently, two parameters can be set: PRT_VDP_SCHEDULING and PRT_SVG_TRACING.

```
void prt_vsa_config_set(
    prt_vsa_t *vsa,
    prt_config_param_t param,
    prt_config_value_t value)
```

PRT_VDP_SCHEDULING can take two values: PRT_VDP_SCHEDULING_AGGRESSIVE and PRT_VDP_SCHEDULING_LAZY. With aggressive scheduling, the VSA tries to fire each VDP as long as possible, i.e., as long as there are packets in all of the VDP's active channels. With lazy scheduling, the VSA fires each VDP once and looks for another VDP to fire. By default, the VSA is set to aggressive scheduling.

`PRT_SVG_TRACING` can take two values: `PRT_SVG_TRACING_ON` and `PRT_SVG_TRACING_OFF`. Turning on the SVG tracing activates PULSAR's internal minimum overhead tracing mechanism, which traces CPU execution, GPU execution, and the activity of the communication proxy, and produces the trace in the form of a *Scalable Vector Graphics* (SVG) file. By default, SVG tracing is off.

PULSAR allows the user to set warmup functions for CPUs and GPUs. These functions will be called by each thread and each device after calling `prt_vsa_run()` and before the VSA starts timing its execution. The motivation is to allow the user to accurately time the workload, while excluding the time of various initializations performed by external software components. The warmup function for CPUs can be set by using the `prt_vsa_thread_warmup_func_set()` function. The warmup function for GPUs can be set by using the `prt_vsa_device_warmup_func_set()` function. Both functions take as parameters: the pointer to the VSA, and the pointer to the warmup function.

```
void prt_vsa_thread_warmup_func_set(
    prt_vsa_t *vsa,
    void(*)() func)
```

```
void prt_vsa_device_warmup_func_set(
    prt_vsa_t *vsa,
    void(*)() func)
```

## 3.2 VDP Operation

This section describes actions, which can take place inside the VDP's function, i.e., the function passed to `prt_vdp_new()` (section 3.1.2). The user never calls that function. It is called by the runtime, when packets are available in all active input channels of the VDP.

### 3.2.1 Packet Creation and Deletion

A new data packet can be created by calling the `prt_vdp_packet_new()` function. The function takes as parameters: the pointer to the VDP creating the packet, the size of the data in bytes, and the pointer to the packet's data payload. If a NULL pointer is passed, the function allocates new memory for the data. The packet's data is accessible through the field `data`.

```
prt_packet_t *prt_vdp_packet_new(
    prt_vdp_t *vdp,
    size_t size,
    void *data)
```

An auxiliary function `prt_vdp_packet_new_host_to_device()` is provided to simplify the common case in GPU programming, when GPU data needs to be initialized by a CPU. The function takes as parameters: the pointer to the VDP creating the packet, the size of the data in bytes, and the pointer to the data. The pointer cannot be NULL and has to reside in the host memory. The function transfers the packet to device memory. After this call, the data cannot be accessed any more by CPU code, and has to be accessed by GPU code.

```
prt_packet_t* prt_vdp_packet_new_host_to_device(
    prt_vdp_t *vdp,
    size_t size,
    void *data)
```

A packet can be released by the VDP by calling the `prt_vdp_packet_release()` function. The function takes as parameters: the VDP releasing the packet and the packet. The runtime will keep the packet and its data around, until it completes all pending operations associated with the packet. However, the packet and its data should not be accessed after the release operation.

```
void prt_vdp_packet_release(
    prt_vdp_t *vdp,
    prt_packet_t *packet)
```

### 3.2.2 Packet Reception and Ejection

A packets can be received by the VDP by calling the `prt_vdp_channel_pop()` function. The function takes as parameters: the VDP receiving the packet, and the channel number, and returns a pointer to the packet received from that channel. After receiving the packet, the packet's data can be accessed through the `data` field.

```
prt_packet_t *prt_vdp_channel_pop(
    prt_vdp_t *vdp,
    int channel_num)
```

A packet can be sent by the VDP by calling the `prt_vdp_channel_push()` function. The function takes as parameters: the VDP sending the packet, the channel number, and the packet. The packet is still available to the VDP after calling that function, and can be used, and repeatedly sent, until it is deleted (section 3.2.1).

```
void prt_vdp_channel_push(
    prt_vdp_t *vdp,
    int channel_num,
    prt_packet_t *packet)
```

### 3.2.3 Channel Deactivation and Reactivation

A channel can be deactivated by the VDP by calling the `prt_vdp_channel_off()` function. The function takes as parameters: the VDP deactivating the channel and the channel number. This indicates to the runtime that it can schedule the VDP (call the VDP function) without checking if there are packets in that channel. The VDP should not attempt to read packets from an inactive channel.

```
void prt_vdp_channel_off(
    prt_vdp_t *vdp,
    int channel_num)
```

A channel can be reactivated by the VDP by calling the `prt_vdp_channel_on()` function. The function takes as parameters: the VDP reactivating the channel and the channel number. This indicates to the runtime that it cannot schedule the VDP (call the VDP function) without if there are no incoming packets in that channel. By default, channels are active.

```
void prt_vdp_channel_on(
    prt_vdp_t *vdp,
    int channel_num)
```

## 3.3 Handling of Tuples

A new tuple can be created by calling the variadic function `prt_tuple_new()`. The function takes as parameters: the length of the tuple, followed by a variable-length list of integer coefficients. A tuple has to have at least one element. There is no upper limit on the number of elements.

```
int* prt_tuple_new  (
    int len,
    int a,
    int b,
    int c,
    ...)
```

In addition, tuples can be created using a set of macros, which contain the length of the tuple in the name and take the elements as the parameters.

```
prt_tuple_new1(a)
prt_tuple_new2(a,b)
prt_tuple_new3(a,b,c)
prt_tuple_new4(a,b,c,d)
prt_tuple_new5(a,b,c,d,e)
prt_tuple_new6(a,b,c,d,e,f)
```

Tuples are dynamically allocated strings or integers, with a the `INT_MAX` constant at the end, serving as the termination symbol. As such, tuples can be free by calling the C standard library `free()` function. However, tuples should not be freed after passing them to the PULSAR runtime. The runtime will free all such tuples during its operation or at the time of calling `prt_vsa_delete()`.

One possible error is passing of `INT_MAX` to the tuple constructors. It will have the silent consequence of indicating the end of the tuple where the used did not intend it. Another possible error is the use of statically allocated tuples, i.e., tuples created on the stack. The runtime will most likely crash when trying to deallocate the memory occupied by such tuples.

# Bibliography

[1] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings*, pages 256–282. Society for Industrial and Applied Mathematics, 1978. ISBN: 0898711606.

[2] H. T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, 1982. DOI: 10.1109/MC.1982.1653825.

[3] H. T. Kung. Systolic array. In *Encyclopedia of Computer Science*, pages 1741–1743. John Wiley and Sons Ltd., Chichester, UK, 2003. ISBN: 0470864125.