A Concurrent Real-Time White Paper

2881 Gateway Drive
Pompano Beach, FL 33069
(954) 974-1700
www.concurrent-rt.com

Improving Real-Time Performance with CUDA Persistent Threads (CuPer) on the Jetson TX2

By: Todd Allen

Consulting Software Engineer

March 2018

Overview

Increasingly, developers of real-time software have been exploring the use of graphics processing units (GPUs) with programming models such as CUDA to perform complex parallelizable calculations. However, historically, GPUs have suffered from problems of poor determinism, which hampers their use for real-time applications when constraints on frame duration are sub-millisecond. This has improved in recent years, but remains a problem for applications with very tight frames, possibly even in the neighborhood of 100 µs.

The use of the persistent threads style can improve determinism significantly, making modest-sized workloads viable for such applications. The paper presents a simple CUDA-based API for this programming style, documents timing results using this API, and makes suggestions on how best to utilize it. All results are based on the Jetson TX2 platform using the RedHawk Linux RTOS with CUDA 8.0.

Real-Time Requirements

The primary requirements for hard real-time software are twofold:

- High Performance
- Determinism

The requirement for high performance is shared with most software. But the requirement for determinism is a particular requirement for real-time software. For any particular computation, the typical measures for these two requirements are ***average time*** and ***jitter***, respectively. Average time is a straightforward computation. In a large algorithm, this often would be measured by its inverse, throughput. In real-time code, the computations frequently are kept deliberately small, and so average time is the more useful value. This often is the easier metric to optimize in a real-time system! In this paper, the *jitter* for a computation is the difference between its ***maximum time*** and its *average time*.

The determination of acceptable values for average time and jitter depend on the duration of the real-time frame, which is the inverse of the frequency of the application. For a soft real-time system, common values might be 60 Hz with a frame duration of 16.666 ms, or 100 Hz with a frame duration of 10 ms. For such applications, the determinism of the standard CUDA launch/synchronize approach probably is sufficient. For a hard real-time system, common values may be anywhere between 1 KHz with a frame duration of 1000 µs to 10 KHz with a

frame duration of 100 µs. For these systems, and especially as the frame durations become shorter and shorter, the average time and jitter must be minimized aggressively.

It also is important to consider that the CUDA computation almost certainly will not be the entirety of the total computation performed within the frame. The portion of the frame that can be dedicated to a CUDA computation varies by the application.

To achieve these goals, programming methods that otherwise might be considered undesirable are considered perfectly valid. A few examples are:

- Use of busy waits
- Forcing system resources to remain completely unused when not needed by the real-time application
- Disabling of power management

CUDA Architecture

Before learning the persistent threads programming style, it first is necessary to understand some key concepts of the CUDA architecture.

In the CUDA programming model, the smallest parallel computation component is the ***thread***. Threads are organized into thread blocks, or simply ***blocks***, of user-specified size. Threads within a block have several mechanisms for fast communication and synchronization between each other. Blocks are expected to function independently of each other. The blocks are further organized into a ***grid***, which encompasses the entirety of the parallel computation. The source and object code that is executed in each of these threads, blocks and grid is called a CUDA ***kernel***. (Despite having the same name, this has no relationship to the Linux kernel.)

In the CUDA hardware model, the smallest computation component is the ***lane***. Lanes are organized into groups called ***warps***, of a size fixed in hardware (32 in all CUDA architectures). At this level of the architecture, the warp functions as a variation of the classic SIMD architecture, with the warp interpreting instructions and applying them in parallel to all lanes within the warp. However, the CUDA architecture supports divergence of threads resulting from data dependent execution flow – at a cost of reduced parallelism – and so the architecture instead is called SIMT.

Each thread is mapped to a single lane. Typically – but not always – a block will comprise more threads than there are lanes within a warp, so the CUDA hardware subdivides the block into multiple warps. In the event that a block is smaller than a warp, or a block is not an integer multiple of the warp size, then some lanes will be inactive.

Above the hierarchical layer of the warp is the streaming multiprocessor, or ***SM***. The SM's operate mostly independently of each other. Blocks are mapped to individual SM's by the CUDA hardware and device driver. Finally, the collection of all SM's is the ***device***.

An SM can support more warps than are capable of executing simultaneously on that SM at one time. Take the Jetson TX2 as an example. This architecture has 2 SM's. Each SM is capable of executing 2 warps simultaneously. Each warp contains 32 lanes. So for the whole device, there are 2*2 = 4 warps, or 2*2*32 = 128 lanes capable of executing simultaneously. However, the device supports 2*64*32 = 4096 ***resident threads***, or 2*64 = 128 ***resident warps*** at once. The reason is its handling of stalls. If an actively executing warp performs one of certain operations that will require more than a cycle to complete – typically a global memory load – that warp is stalled and removed from active execution. On the next cycle, a different warp is executed instead, if one is available. The SM is exploiting the additional parallelism supplied by the collection of resident warps to hide the stalls. It should be noted that the context switches are handled entirely within the CUDA hardware, and can happen from one cycle to the next without performance penalty.

CUDA Kernel Launch/Synchronize

The traditional approach to performing work on the CUDA GPU is first to write a CUDA kernel using CUDA C, which actually is a variant of C++. The code to be executed on the CUDA device is identified by a CUDA function qualifier, `__global__`. The CPU copies data buffers to the device and executes a C-like call with a special <<<>>> syntax to indicate that it is performing a CUDA ***launch***. A CUDA launch is asynchronous, and the CPU will continue to execute afterward. This allows it to perform unrelated work in parallel. When the CPU is ready to wait for the results of the CUDA kernel, it must perform a ***synchronize***, and then it may copy the result buffers back. The source code might have a form like this:

```
__global__ void CudaKernel (float*  A) { … }

void cpuFunction (float* h_A)
{
      …
      cudaMemcpy(d_A, h_A, N);
      CudaKernel<<<blocksPerGrid, threadsPerBlock>>>(d_A);
      cudaDeviceSynchronize();
      cudaMemcpy(h_A, d_A, N);
      …
}
```

However, systems like the Jetson TX2 have a single bank of memory shared between the CPU and GPU, so the memory copies usually are inefficient. A technique called ***zero-copy pinned memory*** may be used instead. This allows the same buffers to be used by both CPU and GPU. The only caveat is that

the memory is **uncached**. This memory is allocated using `cudaHostAlloc` with the `cudaHostAllocMapped` flag. Using this model, the source code for the launch/synchronize might have a form like this:

```
__global__ void CudaKernel (float*  A) { … }

cpuFunction (float* h_A)
{
      …
      cudaHostGetDevicePointer(&d_A, h_A);
      CudaKernel<<<blocksPerGrid, threadsPerBlock>>>(d_A);
      cudaDeviceSynchronize();
      …
}
```

This programming style is simple and powerful. Unfortunately, the launch and synchronize operations have a level of determinism unacceptable for many hard real-time applications.

Persistent Threads

The persistent threads model avoids these determinism problems by launching a CUDA kernel only once, at the start of the application, and causing it to run until the application ends.

There are some drawbacks to this approach:

- It is not possible to launch heterogeneous kernels throughout the application. The kernel is running at all times. However, it is possible to use a switch to select from multiple predefined operations. This will be covered in Heterogeneous GPU Behavior on page 26.
- The kernel must perform a busy wait when waiting for a workload from the CPU. In hard real-time systems, this is not an uncommon solution, but it still is worth noting. It is likely to consume more power than allowing the CUDA GPU to become idle.
- Similarly, the CPU must perform a busy wait when waiting for completion of the GPU.
- The number of blocks and threads used by the kernel is limited to the number of resident threads supported by the device. (This is 4096 on the Jetson TX2.)

The persistent threads concept is very broad and allows for multiple methods of coordination between CPU and GPU, and for allocating work on the GPU[1]. The method described here

[1] Kshitij Gupta, Jeff A. Stuart, John D. Owens. "A study of Persistent Threads Style GPU Programming for GPGPU Workloads" in Innovative Parallel Computing, 2012, pp. 1-14, IEEE, 2012.

closely mirrors that of the launch/synchronization method. It is intentionally simple to provide the best determinism.

Without the explicit launch to start a workload and synchronization to detect when the workload has been completed, other synchronization primitives are needed. It is possible to implement this manually. However, the RedHawk CUDA Persistent Threads (CuPer) API provides a simple API to abstract the primitive operations.

RedHawk CUDA Persistent Threads (CuPer) API

The standard interfaces are provided in the `<cuper.h>` header file. All elements are declared within the `Cuper::Std` namespace. There are three classes defined therein: `Cpu`, `Cuda1Block`, and `CudaMultiBlock`. An object of the `Cpu` class is created in CPU source code. A typical usage would have a form similar to this:

```
void cpuFunction (…)
{
      Cuper::Std::Cpu  p;
      cudaHostGetDevicePointer(&d_A, h_A);
      Persistent<<<blocksPerGrid, threadsPerBlock>>>(p.token(), d_A);
      for (…) {
            … initialize h_A …
            p.startCuda();
            … possibly do unrelated CPU work …
            p.waitForCuda();
            … use results in h_A …
      }
      p.terminateCuda();
}
```

The CUDA kernel in this example is called `Persistent`. It is launched only once before entering the main loop. Any buffer(s) that will be used to pass user data back and forth during normal operation must be specified at that time. In addition, the `Cuper::Std::Cpu` object provides a `token()`, the value of which also must be passed.

The object, `p`, of the `Cuper::Std::Cpu` class is used to control the execution of workloads within the CUDA kernel. Within the main loop, `p.startCuda` informs the CUDA GPU that the input buffers are prepared and that it should begin performing its workload. This is analogous to a CUDA kernel launch. `p.waitForCuda` causes the CPU to wait for the work on the GPU to be completed. This is analogous to a CUDA synchronize.

If it is desired for the main loop ever to exit, `p.terminateCuda` may be called to request that.

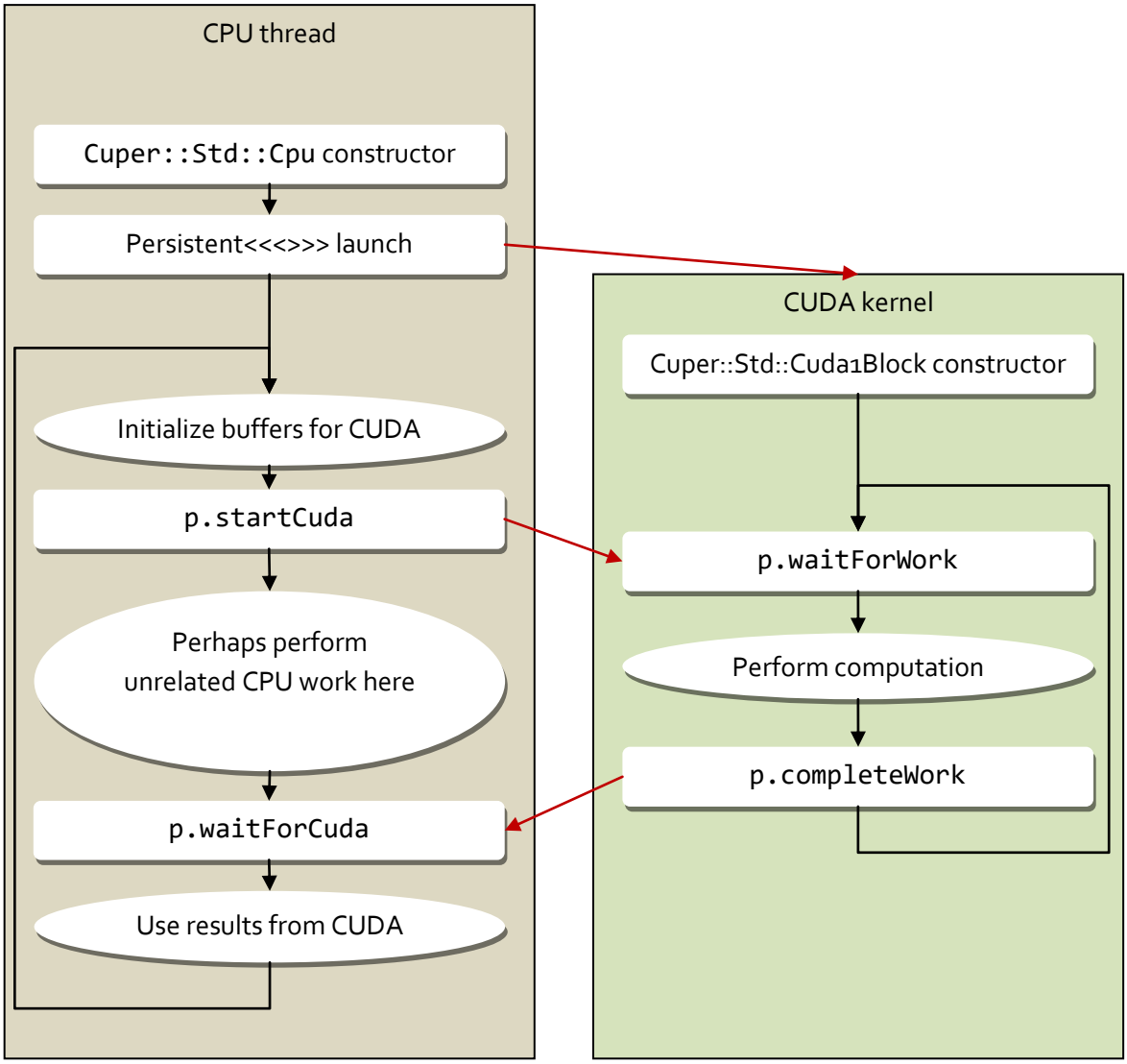
Meanwhile, the `Persistent` kernel in this example might have a form similar to this:

```
__global__ void Persistent (Cuper::Std::Token  token, float*  A)
{
      Cuper::Std::Cuda1Block  p(token);
      for (…) {
            p.waitForWork();
            if (p.isTerminated()) break;
            … perform workload …
            p.completeWork();
      }
}
```

Upon entry to the CUDA kernel, it creates object `p`, a `Cuper::Std::Cuda1Block`, and associates it with its `Cuper::Std::Cpu` counterpart using the `token` value passed from the CPU. This object receives commands from the `Cpu` counterpart and coordinates execution within the CUDA kernel. The kernel calls `p.waitForWork`, which causes the CUDA GPU to wait for a workload from the CPU. Upon return from that, it is safe to execute the workload with consistent user buffers. Once the computation is complete, it calls `p.completeWork` to indicate this to the CPU.

In addition, if the ability to terminate the kernel is desired, a call to `p.isTerminated` can be made to determine whether or not the CPU has requested termination.

The work flow can be visualized with this diagram:



The diagonal arrows show that:

- The `Persistent<<<>>>` launch starts the CUDA kernel
- `p.startCuda` releases the blocked `p.waitForWork`
- `p.completeWork` releases a blocked `p.waitForCuda`

This example shows the use of the `Cuda1Block` class. It is suitable for any CUDA workload which can be performed with only `1` block in the grid. As will be covered in Large Workloads on page 19, this can even include workloads which are arbitrarily large. The `1` block approach often is more efficient than multi-block solutions. However, if multiple blocks are required, `Cuda1Block` can be replaced with `CudaMultiBlock`.

One of the major goals of this approach is the avoidance of CUDA library calls within the main loop, including `cudaMemcpy`. So, zero-copy pinned memory is used in all cases using CuPer.

Performance

To measure the performance of CuPer persistent threads, there are a number of aspects to be considered:

- CuPer API overhead (i.e. the synchronization and fence times)
- GPU Memory I/O
- CPU Memory I/O
- Algorithm (beyond the scope of this paper, but some examples are provided in 1-Block Workloads on page 17)

Note, however, that to calculate the time for a complete workload, it is not a simple matter of adding each of these times together. Because of CUDA's use of resident warps to hide stalls, the times from many of these aspects will be intermingled.

Configuration using RedHawk Linux

The usual collection of real-time features is employed to improve determinism:

- Use of `SCHED_FIFO` scheduler
- Use of real-time priority: 80
- Binding process to a specific CPU: 3
- Shielding of CPU 3 from all processes, interrupts, and ltmr interrupts
  (this is a features specific to RedHawk Linux)
- The timing for the first iteration of every test is discarded, because it is expected to take overly long, while loading various text pages and data objects into caches. Timings begin after this stabilization occurs.

In addition, a couple specific features are used:

- The following Jetson TX2 clocks are set to their maximum allowed values (as is the system fan):
  - CPU
  - GPU (`gp10b`)
  - Memory (`emc`)
- The X server (`Xorg` / `lightdm`) was disabled.
- For the cases involving CUDA launch/synchronization, a call is made to `cudaSetDeviceFlags` with `cudaDeviceScheduleSpin`. This flag is unnecessary with the persistent threads approach, as it would affect only termination.

To simulate non-real-time overhead, the open source `stress-1.0.4` package was used with 5 CPU workers, 5 VM workers, 5 I/O workers, and 5 HDD workers (i.e. `stress -c 5 -m 5 -i 5 -d 5`).

In all cases, timings were performed for at least 2 hours to ensure that average values stabilized and to have reasonable confidence in the maximum values. For certain noted CuPer persistent threads cases, runs were significantly longer to gain even higher confidence in the maximum values.

API Overhead

The first performance tests are measuring only the overhead of the CuPer mechanisms in isolation. For this purpose, the workload is Nil. For the CUDA launch/synchronization cases, it is a CUDA kernel that does nothing. The timing is performed from the CPU using `clock_gettime` with the `CLOCK_REALTIME` clock, and measures the duration from just before the CUDA kernel launch to just after the `cudaDeviceSynchronize` completes.

For the persistent threads cases, it is a kernel that has a main loop that performs only the `p.waitForWork`, `p.isTerminated`, and `p.completeWork` calls, with no actual workload. The timing again is performed from the CPU and measures the duration from just before the `p.startCuda` to just after the `p.waitForCuda` completes.

The first results cover only 1-block cases, with a range of values for `threadsPerBlock`. On all current CUDA architectures, the maximum number of threads per block allowed is 1024:

	CuPer persistent threads			launch/sync		
Threads	Average	Max	Jitter	Average	Max	Jitter
1	2.4 µs	6.6 µs	4.2 µs	17.7 µs	49.0 µs	31.3 µs
64	2.6 µs	5.8 µs	3.2 µs	17.7 µs	47.7 µs	30.0 µs
128	3.2 µs	6.8 µs	3.6 µs	17.8 µs	45.9 µs	28.1 µs
256	4.4 µs	7.6 µs	3.2 µs	18.0 µs	47.2 µs	29.2 µs
512	6.9 µs	11.3 µs	4.4 µs	17.8 µs	46.8 µs	29.0 µs
1024	11.8 µs	16.9 µs	5.1 µs	17.9 µs	45.5 µs	27.6 µs [2]

TABLE 1: API OVERHEAD, 1 BLOCK (NIL)[2]

[2] The 1024-thread CuPer timing was performed for at least 24 hours.

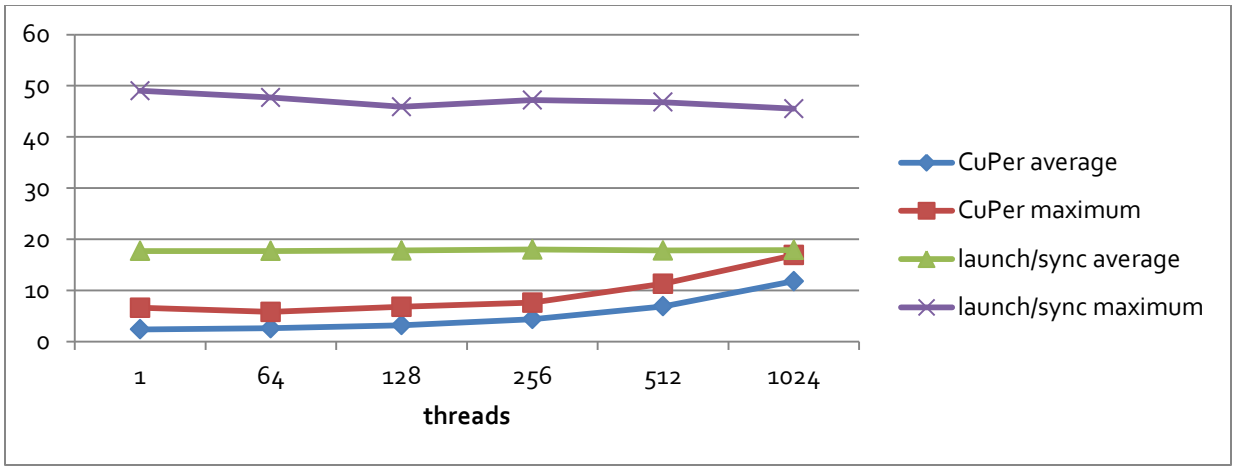


FIGURE 1: API OVERHEAD, 1 BLOCK (NIL)

As can be seen, the CUDA launch/synchronization maximum times essentially are fixed, whereas the CuPer `startCuda`/`waitForCuda` times scale with the number of threads in the block. However, the launch/synchronization times suffer from substantial jitter, which is greatly reduced in the CuPer times.

Moving beyond the 1-block cases, further timings were performed for 2- and 4- block configurations. These values, along with the 1-block values above are presented here:

Total		CuPer persistent threads			launch/sync zero-copy		
Threads	**Blocks**	Average	Max	Jitter	Average	Max	Jitter
128	**1**	3.2 µs	6.8 µs	3.6 µs	17.8 µs	45.9 µs	28.1 µs
	2	4.2 µs	19.9 µs	15.7 µs			
	4	6.7 µs	24.2 µs	17.5 µs			
256	**1**	4.4 µs	7.6 µs	3.2 µs	18.0 µs	47.2 µs	29.2 µs
	2	5.4 µs	19.3 µs	13.9 µs			
	4	8.3 µs	25.7 µs	17.4 µs			
512	**1**	6.9 µs	10.7 µs	3.8 µs	17.8 µs	46.8 µs	29.0 µs
	2	7.8 µs	21.7 µs	13.9 µs			
	4	10.4 µs	27.6 µs	17.2 µs			
1024	**1**	11.8 µs	16.0 µs	4.2 µs	17.6 µs	45.4 µs	27.8 µs
	2	12.8 µs	26.1 µs	13.3 µs			
	4	15.4 µs	25.8 µs	10.4 µs			
2048	**2**	22.6 µs	39.8 µs	17.2 µs	17.9 µs	45.5 µs	27.6 µs
	4	25.0 µs	30.3 µs	5.3 µs			
4096	**4**	44.8 µs	52.2 µs	7.4 µs	17.9 µs	45.0 µs	27.1 µs

TABLE 2: API OVERHEAD, MULTI-BLOCK (NIL)

These timings demonstrate that using multiple blocks, even if the total number of threads remains constant, imposes a significant overhead penalty in CuPer. A large part of this is because of a necessary grid-wide synchronization in `CuPer::Std::CudaMultiBlock::completeWork`. In fact, if using the full set of resident threads (4096), the CuPer average performance actually degrades to the point where the launch/synchronize approach outperforms it. So, for small workloads, it is recommended to use a 1-block configuration and `CuPer::Std::Cuda1Block`. For larger workloads, it is possible to perform the entire workload within 1 block. But the best results depend on the particular algorithm. This is discussed in **Large Workloads** on page 19.

GPU Memory I/O

Performing timings in this parallel CUDA architecture is not a simple matter of adding overhead times to I/O times and to computation times, because portions of the CuPer processing will overlap with those of the computation and memory I/O. So it is necessary to perform more timings of increasing completeness. The next level of completeness incorporates the additional time for memory I/O in the CUDA GPU. As mentioned previously, zero-copy memory is **uncached**. Yet the GPU ameliorates this by exploiting parallelism to hide memory stalls.

To begin with, consider what is perhaps the simplest possible workload: vector increment. This code takes an array of floating-point values, and increments each of them by 1.0, returning the results in the same array used to pass the initial values. This requires only 1 buffer, and the workload is perfectly parallelizable. The computation time is negligible, and is just enough to actually require reading from and writing different values to the buffer. So this serves as a measure of the performance of the CuPer API coupled with GPU Memory I/O. The timings measure performance in exactly the same way as for the Nil workload.

For this workload, because there is an actual data buffer involved, the CUDA launch/synchronize timings can be performed using a number of memory approaches: zero-copy pinned memory, the original `cudaMemcpy` approach, and also UVM (also called managed memory). To begin with, the following are 1-block timings:

	Threads	64	128	256	512	1024
CuPer persistent threads	Average	4.0 µs	4.7 µs	6.1 µs	8.5 µs	13.3 µs [3]
	Max	7.6 µs	8.3 µs	9.4 µs	12.2 µs	18.0 µs [3]
	Jitter	3.6 µs	3.6 µs	3.3 µs	3.7 µs	4.7 µs [3]
launch/sync zero-copy	Average	19.6 µs	19.4 µs	20.2 µs	20.8 µs	22.0 µs
	Max	48.5 µs	45.6 µs	50.8 µs	49.6 µs	52.5 µs
	Jitter	28.9 µs	26.2 µs	30.6 µs	28.8 µs	30.5 µs
launch/sync cudaMemcpy	Average	53.2 µs	51.8 µs	54.8 µs	56.1 µs	55.3 µs
	Max	100.3 µs	102.0 µs	98.9 µs	113.6 µs	92.5 µs
	Jitter	47.1 µs	50.2 µs	44.1 µs	57.5 µs	37.2 µs
launch/sync UVM	Average	39.1 µs	39.2 µs	39.3 µs	40.1 µs	41.6 µs
	Max	80.4 µs	140.7 µs	81.5 µs	81.8 µs	125.4 µs
	Jitter	41.3 µs	101.5 µs	42.2 µs	41.7 µs	83.8 µs

TABLE 3: API OVERHEAD + GPU MEMORY I/O, 1 BLOCK (VECTOR INCREMENT) [3]

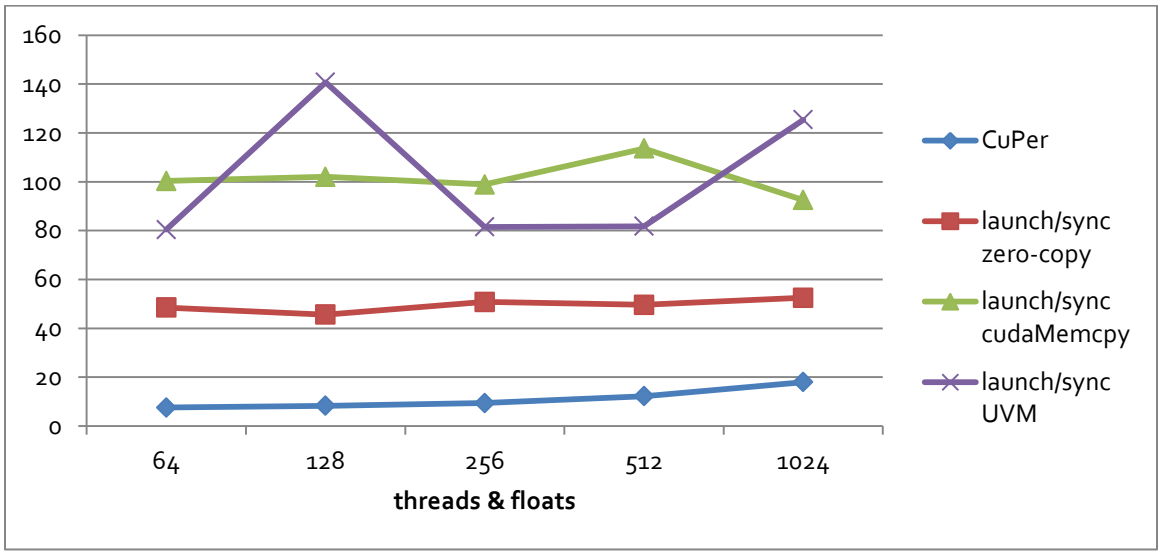


FIGURE 2: API OVERHEAD + GPU MEMORY I/O, 1 BLOCK, MAXIMUM (VECTOR INCREMENT)

These results demonstrate that, when including the GPU memory I/O times, the persistent threads approach still performs well in terms of average time, and very well in terms of jitter.

[3] The 512- and 1024-thread CuPer timings were performed for at least 24 hours.

Regarding UVM, it appears to suffer from long period jitter. The timings here did not all capture instances of this, and so probably do not precisely capture the true maximums. To do so would require days of run time, but that is not the focus of this paper.

CPU Memory I/O

As was previously mentioned, both CuPer and the CUDA launch/synchronize approaches use zero-copy pinned memory. But a consequence of this memory is that it is **uncached**. Measurement of the GPU memory I/O is performed automatically as part of any timing, because the GPU loads and stores occur within the timed region. However, CPU memory I/O does not. It generally is interspersed with initialization code of varying complexity, and with interpretation code of differently varying complexity. Therefore it is difficult to isolate. Yet, it still is worthwhile to provide some guidance on these times.

The approximation used here is `memcpy`. The initial values are written first to staging buffers, and then copied via `memcpy` to the actual input buffers shared between the CPU and CUDA. Likewise, after completion of the workload, the output buffers are copied via `memcpy` to staging buffers, and then they are validated from the staging buffers. This is **not** meant to suggest that using `memcpy` is appropriate in user code. It is just present to represent the stores and loads that user could would perform naturally. That said, there may be certain memory usage patterns where `memcpy` calls for some or all buffers might be beneficial.

Timings are performed using the trivial vector increment workload. The timing is performed from just before the start of the `memcpy` call until just after it returns. One timing is performed for storing to the input buffers, and another for loading from the output buffers.

Timings are performed on these types of memory:

1. Stores and loads from dummy buffers allocated via `malloc`. The purpose of this is to establish a baseline for stores and loads to an appropriate amount of memory which is cached normally.
2. Stores and loads to a buffer allocated via `malloc` and referenced via `cudaMemcpy` in a CUDA launch/synchronize implementation which does not use zero-copy pinned memory.
3. Stores and loads to UVM memory allocated via `cudaMallocManaged` used with a CUDA launch/synchronize approach.
4. Stores and loads to a zero-copy pinned memory buffer used with a CUDA launch/synchronize approach.
5. Stores and loads to a zero-copy pinned memory buffer used with CuPer persistent threads.

As one would expect, the timings scale with the size of the buffers, so a few buffer sizes are presented:

Size	Memory type	Store to Input buffer			Load from Output buffer		
		Average	Max	Jitter	Average	Max	Jitter
64	Ordinary memory	.1 µs	.8 µs	.7 µs	.1 µs	1.0 µs	.9 µs
	cudaMemcpy buffer	.1 µs	1.7 µs	1.6 µs	.1 µs	2.1 µs	2.0 µs
	cuda UVM buffer	.1 µs	1.7 µs	1.6 µs	2.4 µs	14.0 µs	11.6 µs
	Zero-copy: launch/synchronize	.1 µs	1.5 µs	1.4 µs	.5 µs	5.0 µs	4.5 µs
	Zero-copy: CuPer	.1 µs	1.5 µs	1.4 µs	.4 µs	4.3 µs	3.9 µs
256	Ordinary memory	.1 µs	1.0 µs	1.0 µs	.1 µs	.8 µs	.7 µs
	cudaMemcpy buffer	.1 µs	1.5 µs	1.4 µs	.2 µs	3.0 µs	2.8 µs
	cuda UVM buffer	.1 µs	1.6 µs	1.5 µs	2.9 µs	18.0 µs	15.1 µs
	Zero-copy: launch/synchronize	.7 µs	3.6 µs	2.9 µs	1.7 µs	8.8 µs	7.1 µs
	Zero-copy: CuPer	.7 µs	3.1 µs	2.4 µs	1.5 µs	7.5 µs	6.0 µs
1024	Ordinary memory	.3 µs	1.4 µs	1.1 µs	.3 µs	1.7 µs	1.4 µs
	cudaMemcpy buffer	.3 µs	1.7 µs	1.4 µs	.6 µs	5.3 µs	4.7 µs
	cuda UVM buffer	.4 µs	3.0 µs	2.6 µs	3.9 µs	15.9 µs	12.0 µs
	Zero-copy: launch/synchronize	1.3 µs	9.4 µs	8.1 µs	6.1 µs	16.0 µs	9.9 µs
	Zero-copy: CuPer	1.3 µs	7.2 µs	5.9 µs	6.0 µs	14.2 µs	8.2 µs

TABLE 4: CPU MEMORY I/O

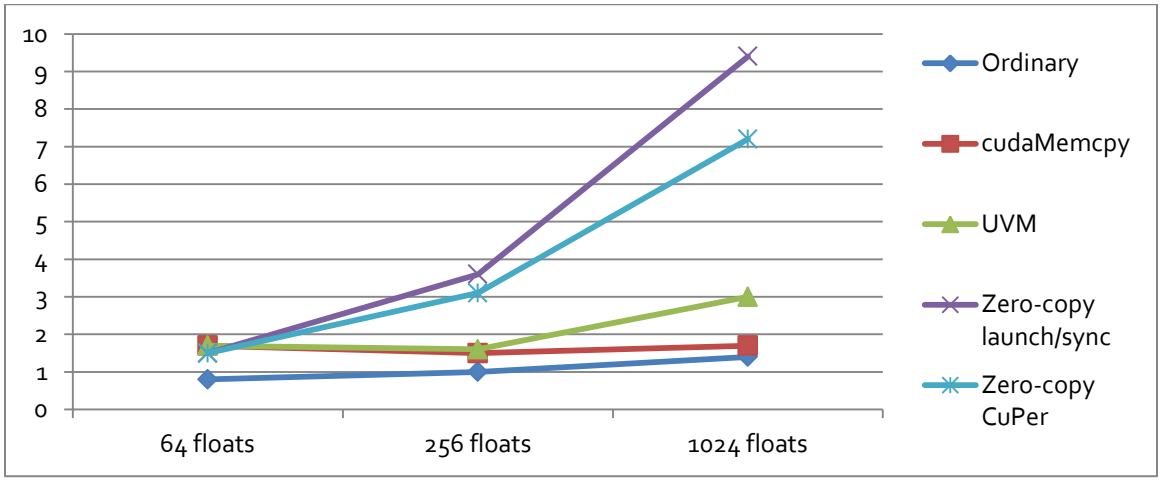


FIGURE 3: CPU MEMORY I/O: STORE TO INPUT BUFFER, MAXIMUM

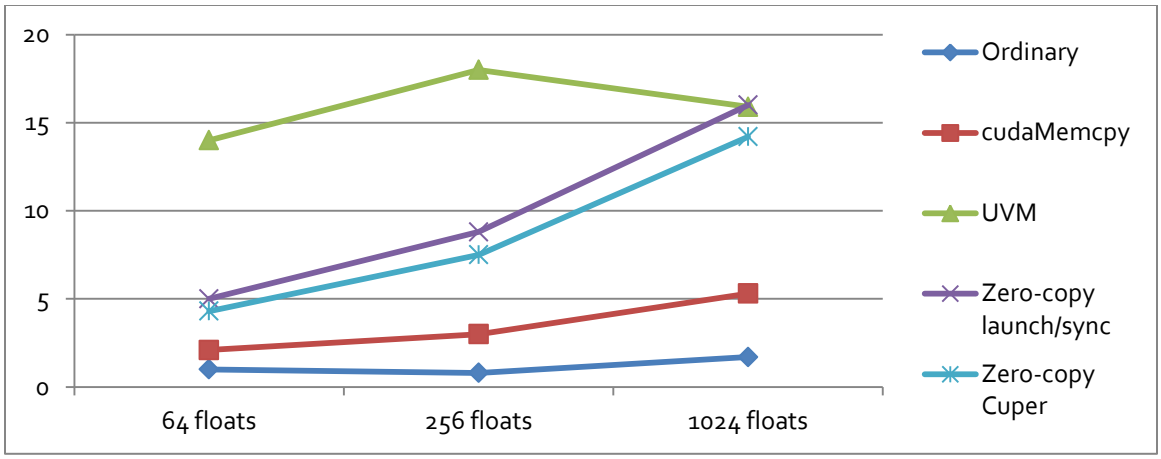


FIGURE 4: CPU MEMORY I/O: LOAD FROM OUTPUT BUFFER, MAXIMUM

Not surprisingly, the two tests that used memory allocated via `malloc` fared well and similarly to each other. The UVM test was inconsistent, performing well on storing inputs, but poorly on loading outputs (on the first memory access only, though). The two tests that used the uncached zero-copy pinned memory fared less well and similarly to each other. In a user application, it is likely that these time costs will be spread throughout the code that initializes the CUDA input buffers and in the code that reads the results from the CUDA output buffers.

Although this additional time cost is an artefact of using zero-copy memory, and not caused by the persistent threads approach per se, it still is important to understand that it is present and to account for it in when determining the overall performance of a real-time application.

Because of the problem of characterizing and isolating the CPU Memory I/O, and because nearly all subsequent timings will suffer similarly, subsequent timings do not include it. In a couple cases where `cudaMemcpy` times are measured, they are noted as not suffering from significant additional CPU Memory I/O time.

CUDA Dynamic Parallelism

There is one other approach to using persistent kernels that differs substantially from the previous examples. In this approach, the persistent kernel executes only a single thread. After that thread returns from `p.waitForWork`, it does not actually perform the workload itself at all. Instead, it launches another kernel using CUDA dynamic parallelism (CDP), waits for that kernel to complete, and then calls `p.completeWork`.

This is very similar to the traditional CUDA launch/synchronize programming model. The major difference is that it moves the launch and synchronize operations to the CUDA GPU. Unfortunately, it suffers from long period jitter spikes. So for 1-block workloads, it is not competitive with the persistent threads approaches described above. Note that it takes this approach a bit longer than normal before

its timing behavior stabilizes. In results presented in the next section, the timings for the first **2** iterations are discarded.

Code to use this approach would have a form like the following:

```
__global__ void Algorithm (float*  Result, float*  A) { … }

__global__ void Persistent (Cuper::Std::Token  token,
                            float*  Result, float*  A)
{
      for (…) {
            p.waitForWork();
            Algorithm<<<16, 1024>>>(Result, A);
            cudaDeviceSynchronize();
            p.completeWork();
      }
}

void cpuFunction (…)
{
      Persistent<<<1, 1>>>(p.token(), d_Result, d_A);
      for (…) {
            …
            p.startCuda();
            p.waitForCuda();
            …
      }
}
```

1-Block Workloads

A couple common workloads are presented to show their performance with 1-block implementations:

- Matrix multiplication of a 32x32 matrix, which fits neatly into 32*32 = 1024 threads in 1 block. This is a standard shared-memory implementation.
  Note that, in the launch/synchronize cases, its takes a **very** long time for the performance to stabilize. So, for those cases, instead of discarding the first iteration, they discard the first **512** iterations.
- Vector sum of a 1024-element vector, which is an example of a reduction algorithm. This is a standard algorithm using successively smaller strides, but which switches to using warp-level primitives when the strides are small enough to fit within the 32 lanes of a warp.

The results of these timings are as follows:

	Average	Max	Jitter
CuPer persistent threads	14.6 µs	18.7 µs	4.1 µs [4]
CuPer CDP	22.2 µs	70.0 µs	47.8 µs
launch/sync zero-copy	34.6 µs	78.8 µs	44.2 µs
launch/sync cudaMemcpy	90.5 µs	152.0 µs	61.5 µs [5]
launch/sync UVM	66.4 µs	126.0 µs	59.6 µs

TABLE 5: 32X32 [4]MATRIX MULTIPLICATION + API OVERHEAD + GPU MEMORY I/O[5]

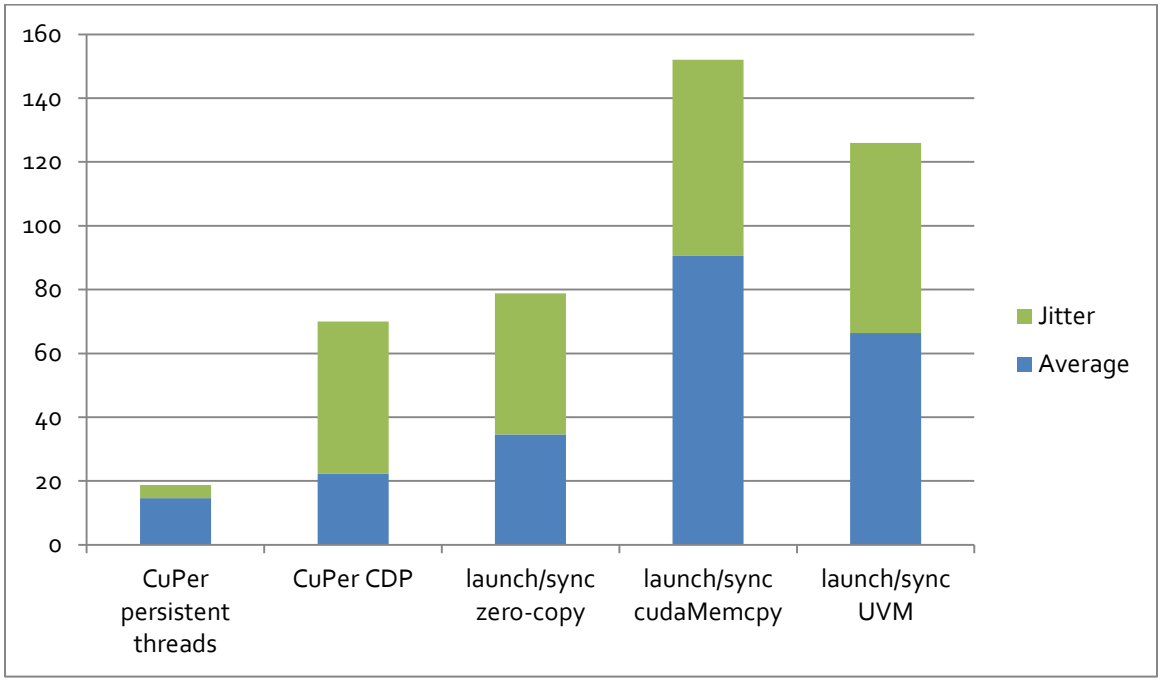


FIGURE 5: 32X32 MATRIX MULTIPLICATION + API OVERHEAD + GPU MEMORY I/O

[4] The CuPer 32x32 Matrix Multiplication timing run was performed for at least 24 hours.
[5] The `cudaMemcpy` approach will not suffer from additional CPU Memory I/O time cost. See CPU Memory I/O on page 14.

	Average	Max	Jitter
CuPer persistent threads	13.4 µs	17.5 µs	4.1 µs [6]
CuPer CDP	19.6 µs	69.1 µs	49.5 µs
launch/sync zero-copy	27.2 µs	61.1 µs	33.9 µs
launch/sync cudaMemcpy	60.9 µs	108.9 µs	48.0 µs [5]
launch/sync UVM	47.0 µs	101.0 µs	54.0 µs

TABLE 6: 1K VECTOR SUM + API OVERHEAD + GPU MEMORY I/O[6]

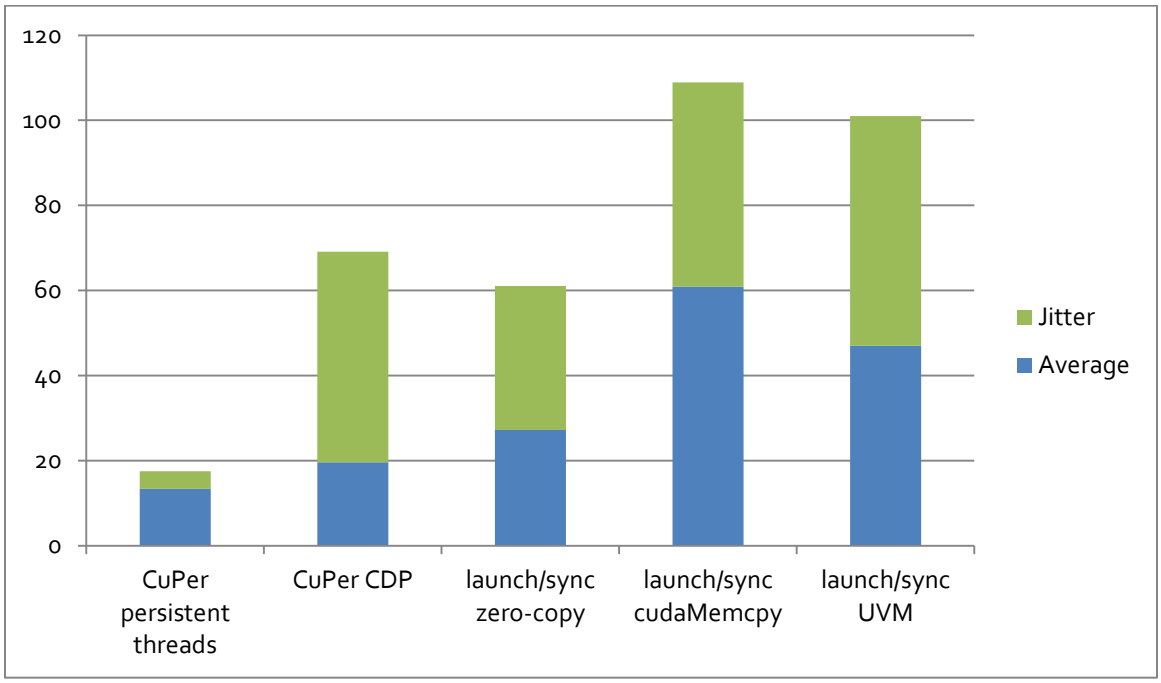


FIGURE 6: 1K VECTOR SUM + API OVERHEAD + GPU MEMORY I/O

As demonstrated by these timings, the CuPer persistent threads approach yields much better performance, in terms of both average time and jitter, than launch/sync approach or the CuPer CDP approach for 1-block workloads. However, do bear in mind that these timings do not include the distributed CPU Memory I/O overhead discussed in **CPU Memory I/O** on page 14.

Large Workloads

1-Block workloads provide good performance that can be integrated into tight real-time frame requirements. However, there will be cases where larger workloads are necessary, and the frame requirements can accommodate them. There are a number of persistent thread techniques that can be employed for larger workloads:

[6] The CuPer 1K Vector Sum timing run was performed for at least 24 hours.

- Multi-block kernel
- Single-block kernel with ***thread groups***
- Hybrid: Multi-block kernel with thread groups
- Single-thread kernel which performs CUDA dynamic parallelism (CDP) launch/synchronize

Multi-block kernel

The multi-block kernel approach has been explored in a few cases earlier. It merely involves configuring the kernel with more than 1 block. However, with the persistent threads approach, there exists a hard upper limit on the number of threads: the maximum number of resident threads, which is 4096 on the Jetson TX2. So, if a block is configured to use its maximum size, 1024, the number of blocks can be no more than 4. So this approach necessarily is limited. Also, it comes with performance penalties, depending on the configuration. Code using this technique would have an arrangement like the following:

```
__global__ void Algorithm (Cuper::Std::Token  token,
                           float*  Result, float*  A)
{
      int  index = blockIdx.x * blockDim.x + threadIdx.x;
      for (…) {
            p.waitForWork();
            …
            p.completeWork();
      }
}

void cpuFunction (…)
{
      Algorithm<<<4, 1024>>>(p.token(), d_Result, d_A);
      for (…) {
            …
            p.startCuda();
            p.waitForCuda();
            …
      }
}
```

Single-block kernel with Thread Groups

Another approach is to use 1 block, but instead partition the workload into multiple ***thread groups***. A thread group is a software concept which replaces the CUDA hardware block concept. And, whereas a

collection of CUDA blocks would be scheduled by the CUDA hardware scheduler, each thread group must be executed manually by the single running block. Although more complex scheduling methods could be implemented, they incur performance overhead, so the simplest way for a real-time application to do this is to iterate over the thread groups. Using thread groups, for a 16K example, the code would have an arrangement like the following:

```
__global__ void Algorithm (Cuper::Std::Token  token,
                           float*  Result, float*  A)
{
      for (…) {
            p.waitForWork();
            for (unsigned int group = 0; group < 16; group++) {
                  int  index = group * blockDim.x + threadIdx.x;
                  …
            }
            p.completeWork();
      }
}

void cpuFunction (…)
{
      Algorithm<<<1, 1024>>>(p.token(), d_Result, d_A);
      for (…) {
            …
            p.startCuda();
            p.waitForCuda();
            …
      }
}
```

Hybrid: Multi-block kernel with Thread Groups

A hybrid approach is one where the persistent kernel is configured with multiple blocks, but those blocks are insufficient to handle the entire workload. So, thread groups are used as well. In this case, each thread group contains all the blocks in the grid. So, for a 16K example, the configuration might be 1024 threads/block * 2 blocks/group * 8 groups = 16K. There are a couple different ways of dividing up the blocks amongst the groups but, using this configuration, the code might have an arrangement like the following:

```
__global__ void Algorithm (Cuper::Std::Token  token,
                           float*  Result, float*  A)
{
      for (…) {
            p.waitForWork();
            for (unsigned int group = 0; group < 8; group++) {
                  int  pseudoBlock = group * gridDim.x + blockIdx.x;
                  int  index = pseudoBlock * blockDim.x + threadIdx.x;
                  …
            }
            p.completeWork();
      }
}

void cpuFunction (…)
{
      Algorithm<<<2, 1024>>>(p.token(), d_Result, d_A);
      for (…) {
            …
            p.startCuda();
            p.waitForCuda();
            …
      }
}
```

Single-thread kernel with CUDA Dynamic Parallelism

Although the use of CUDA dynamic parallelism approach described in **CUDA Dynamic Parallelism** on page 16 was not competitive for 1-block workloads, it can be effective for larger workloads, even though it suffers from long period jitter spikes.

In this case, the CUDA hardware scheduler is responsible for scheduling all the blocks of the child kernel.

Performance of Large Workload approaches

The best of the above approaches differs from workload to workload, and so timings for a few familiar ones are presented.

To begin, consider the trivial vector increment workload, scaled up to 32K elements:

	Average	Max	Jitter
CuPer: Multi-group (1 block)	29.8 µs	34.3 µs	4.5 µs
CuPer: Hybrid: 2 blocks	33.9 µs	38.4 µs	4.5 µs
CuPer: Hybrid: 4 blocks	54.4 µs	59.9 µs	5.5 µs
CuPer: CDP	28.8 µs	81.6 µs	52.8 µs
launch/sync zero-copy	43.9 µs	77.4 µs	33.5 µs

TABLE 7: 32K VECTOR INCREMENT + API OVERHEAD + GPU MEMORY I/O

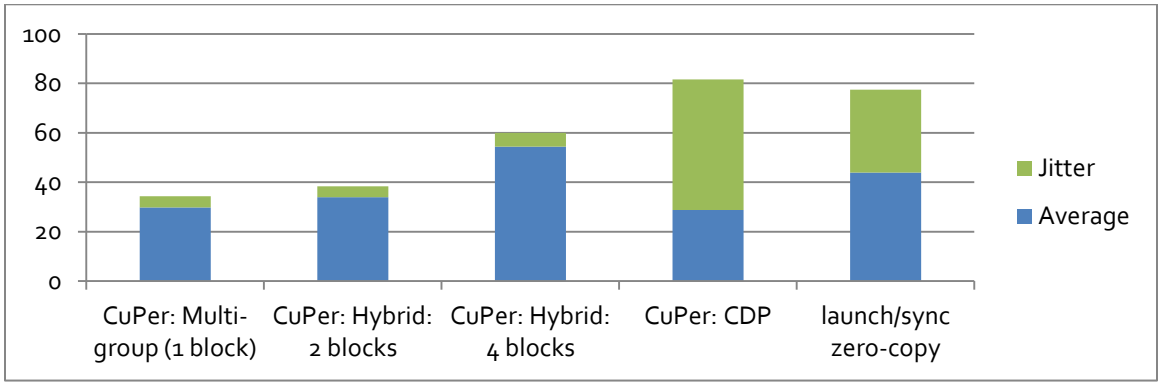


FIGURE 7: 32K VECTOR INCREMENT + API OVERHEAD + GPU MEMORY I/O

The best average performance and determinism in this case comes from the CuPer multi-group (1 block) kernel. The CuPer CDP approach looked quite promising at first with its good average time, but, as mentioned earlier, it suffers from long period jitter spikes.

Next, consider a more significant workload, but one which is well-understood: matrix multiplication. Two examples of this are provided. The first is scaled up to 64x64, which would require 2*2 = 4 blocks of 32*32 = 1024 threads each. The second is scaled up to 128x128, which would require 4*4 = 16 blocks of 32*32 = 1024 threads each. The various techniques described here yield the following timings:

	Average	Max	Jitter
CuPer: Multi-group (1 block)	29.6 µs	33.7 µs	4.1 µs
CuPer: Hybrid: 2 blocks	41.5 µs	54.2 µs	12.7 µs
CuPer: Multi-block (2x2 blocks)	71.4 µs	82.3 µs	10.9 µs
CuPer: CDP	30.6 µs	75.8 µs	45.2 µs
launch/sync zero-copy	54.3 µs	92.3 µs	38.0 µs

TABLE 8: 64X64 MATRIX MULTIPLICATION + API OVERHEAD + GPU MEMORY I/O

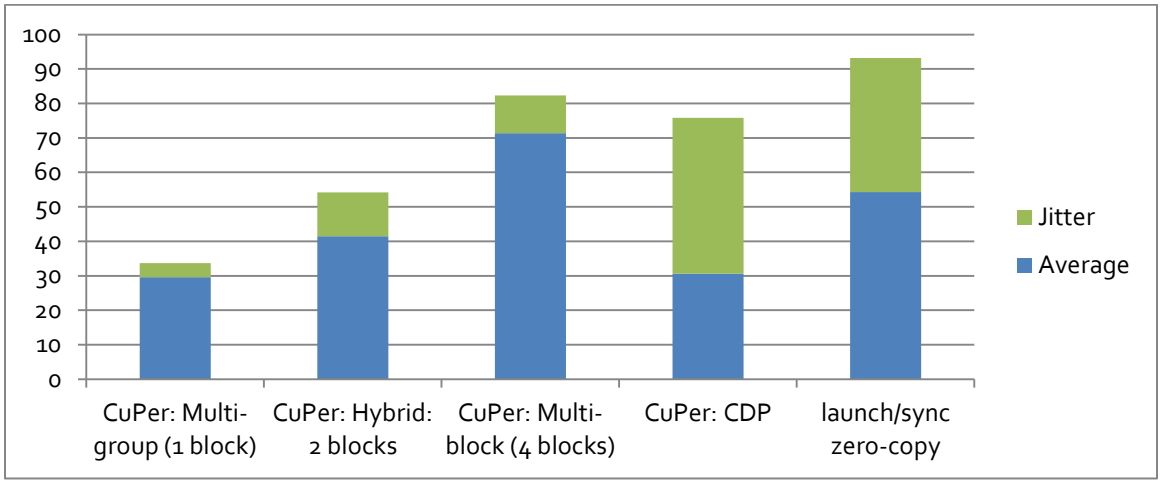


FIGURE 8: 64X64 MATRIX MULTIPLICATION + API OVERHEAD + GPU MEMORY I/O

In the 64x64 case, the CuPer Multi-group (1 block) implementation also performed the best, and did significantly better than either hybrid approach.

	Average	Max	Jitter
CuPer: Multi-group (1 block)	147.2 µs	158.1 µs	10.9 µs
CuPer: Hybrid: 2 blocks	156.6 µs	169.0 µs	12.4 µs
CuPer: Hybrid: 4 blocks	238.0 µs	259.0 µs	21.0 µs
CuPer: CDP	92.8 µs	133.8 µs	41.0 µs
launch/sync zero-copy	361.7 µs	396.8 µs	35.1 µs

TABLE 9: 128X128 MATRIX MULTIPLICATION + API OVERHEAD + GPU MEMORY I/O

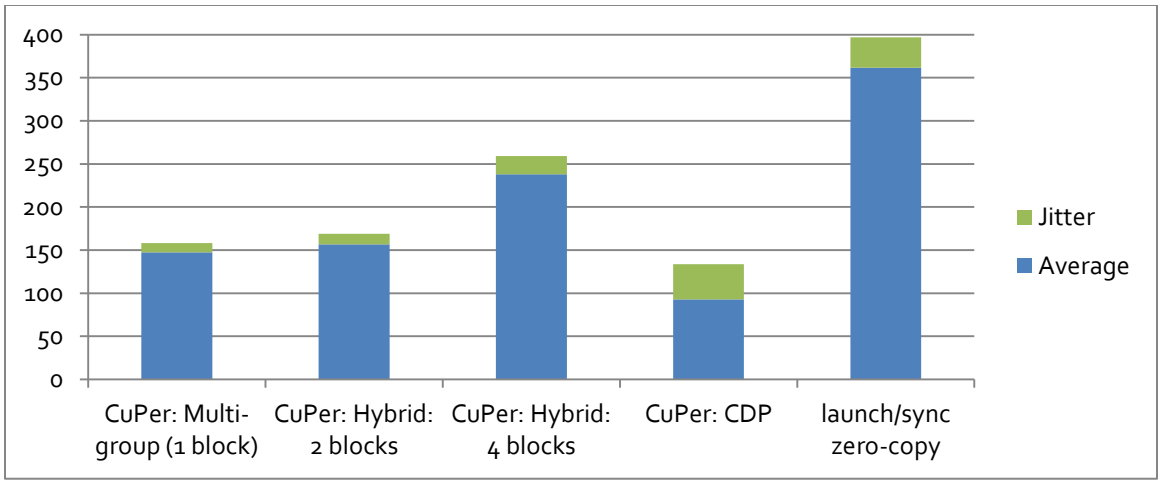


FIGURE 9: 128X128 MATRIX MULTIPLICATION + API OVERHEAD + GPU MEMORY I/O

The 128x128 case is an interesting example because the CUDA dynamic parallelism approach, despite having the worst determinism, has very good average performance, resulting in the lowest maximum observed time!

Finally, consider vector sum, a reduction algorithm. This example is scaled up to 32K elements:

	Average	Max	Jitter
CuPer: Multi-group (1 block)	61.6 µs	66.4 µs	4.8 µs
CuPer: Hybrid: 2 blocks	52.5 µs	57.6 µs	5.1 µs
CuPer: Hybrid: 4 blocks	63.8 µs	70.1 µs	6.3 µs
CuPer: CDP	46.6 µs	90.5 µs	43.9 µs
launch/sync zero-copy	308.3 µs	339.4 µs	31.1 µs

TABLE 10: 32K VECTOR SUM + API OVERHEAD + GPU MEMORY I/O

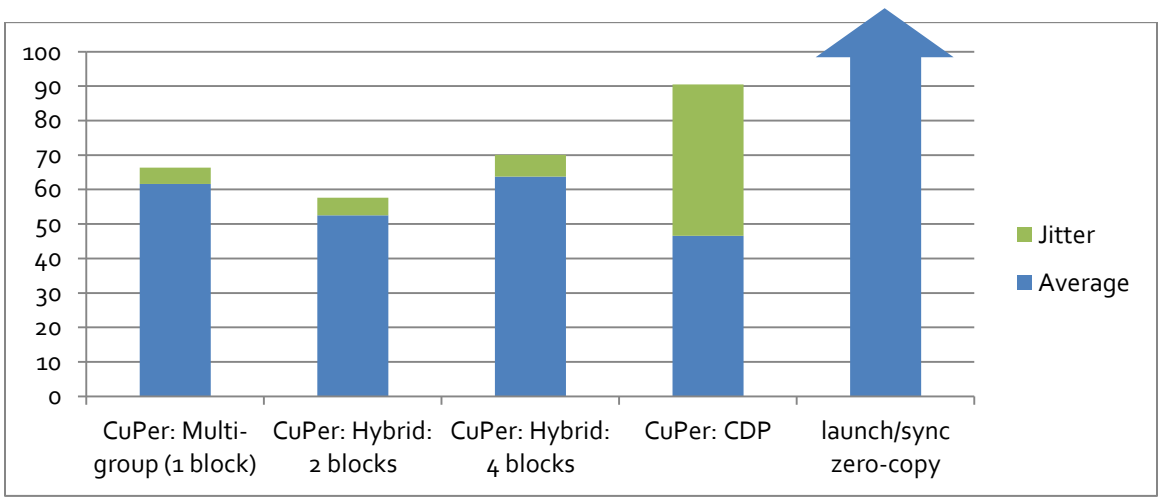


FIGURE 10: 32K VECTOR SUM + API OVERHEAD + GPU MEMORY I/O

In this case, the hybrid 2 block implementation performed the best, beating out both the pure 1-block multi-group implementation, and its 4 block hybrid counterpart. This demonstrates that it is not always obvious in advance which implementation will perform the best. In this case, the hybrid 2-block approach's ability to use the 2nd SM provided enough performance benefit to outweigh the increased overhead and jitter costs of using multiple blocks.

Heterogeneous GPU Behavior

Earlier, it was pointed out that one drawback of the persistent threads approach is that, because a single kernel is launched at application start and runs until application end, it is not possible to launch a variety of heterogeneous kernels. However, it is possible to launch a persistent kernel which is capable of a variety of different behaviors, and to command it to perform a specific behavior. Code to accomplish this could look something like the following:

```
enum Command { MatMul_32x32x32, VecSum_32K };

__global__ void Persistent (Cuper::Std::Token  token,
                            unsigned int*      Command,
                            float* Buf0, float* Buf1, float* Buf2);
{
      for (…) {
            p.waitForWork();
            unsigned int  cmd = *Command;
            switch (cmd) {
            case MatMul_32x32x32:
                  // C = Buf0, A = Buf1, B = Buf2
                  dim3 twoDimIdx(thread.Idx.x & 0x1f, threadIdx.x>>5);
                  matrixMultiply(twoDimIdx, Buf0, Buf1, Buf2);
                  break;
            case VecSum_32K:
                  // Result = Buf0, Vector = Buf1
                  vectorSum(Buf0, Buf1);
                  break;
            }
            p.completeWork();
      }
}
```

```
coid cpuFunction (…)
{
      Persistent<<<1, 1024>>>(p.token(),
                              d_Command, d_Buf0, d_Buf1, d_Buf2);
      for (…) {
            …
            initializeMatrixA(h_Buf1);
            initializeMatrixB(h_Buf2);
            *h_Command = MatMul_32x32x32;
            useMatrixC(h_Buf0);
            p.startCuda();
            p.waitForCuda();
            …
            initializeVector(h_Buf1);
            *h_Command = VecSum_32K;
            p.startCuda();
            p.waitForCuda();
            useResult(h_Buf0[0]);
            …
      }
}
```

Restrictions of this approach are:

- The persistent kernel can have only one block and grid configuration. If a particular function expects something else, glue code must be used to perform the translation. In the above example, `matrixMultiply` expects 2-dimensional thread `threadIdx` coordinates. The `twoDimIdx` translates from the persistent kernel's 1-dimensional coordinates. And then `matrixMultiply` must use that instead of the ordinary `threadIdx`.
- The buffers must be able to accommodate all possible behaviors of the persistent kernel. For example, in the above case, 3 buffers are required, and the size for each is the maximum of the requirements for each behavior. For the above:
    - Bufo: MatMul_32K requires 32*32 = 1024 floats; VecSum_32K requires 1 float.
    - Buf1: MatMul_32K requires 32*32 = 1024 floats; VecSum_32K requires 32K floats.
    - Buf2: MatMul_32K requires 32 floats; VecSum_32K requires none.

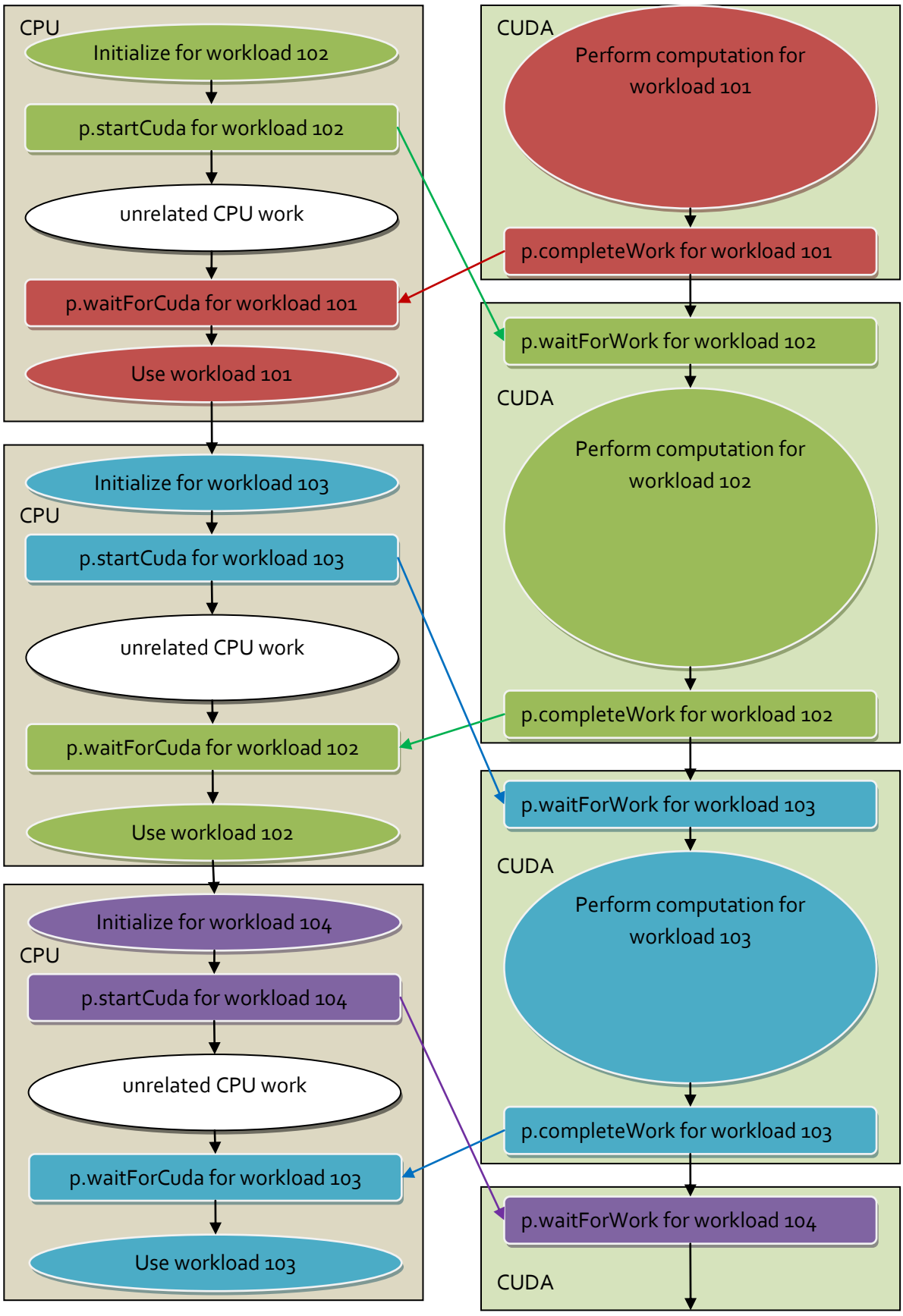
  Alternatively, independent sets of buffers could have been used.
- If any of the behaviors require dynamically-allocated shared memory using the 3rd parameter in the CUDA <<<>>> launch configuration, it must be the maximum required by any of the behaviors.

Double Buffering

Recall from the earlier section that introduced CuPer that it is possible to perform an unrelated CPU workload while performing the GPU work. That extra degree of parallelization is useful. However, the CUDA GPU remains idle when not explicitly running a workload assigned by the CPU. This can be improved upon by double buffering, if a 1-frame delay in response is acceptable to the application.

The idea behind double buffering is that the CPU initially starts one workload on the GPU. Then, for each frame, it starts a workload and then uses the results of the **previous** workload. For example, in the CPU's 101st iteration, it starts workload 101, and uses the results from workload 100. This can be thought of as a 2-stage pipeline.

CuPer provides an alternate interface to support this within the `Cuper::DoubleBuffer` namespace. It closely mirrors the interface under `Cuper::Std`, but with some additional functions. The workload can be visualized with this diagram for the flow near workloads 102 and 103:


CPU
Initialize for workload 102
p.startCuda for workload 102
unrelated CPU work
p.waitForCuda for workload 101
Use workload 101
CUDA
Perform computation for workload 101
p.completeWork for workload 101
p.waitForWork for workload 102
CUDA
Perform computation for workload 102
p.completeWork for workload 102
Initialize for workload 103
CPU
p.startCuda for workload 103
unrelated CPU work
p.waitForCuda for workload 102
Use workload 102
p.waitForWork for workload 103
CUDA
Perform computation for workload 103
p.completeWork for workload 103
Initialize for workload 104
CPU
p.startCuda for workload 104
unrelated CPU work
p.waitForCuda for workload 103
Use workload 103
p.waitForWork for workload 104
CUDA

The diagonal arrows in the diagram demonstrate the flow of work between CPU and GPU. Each workload is assigned a color, so it is possible to see how it progresses through the diagram. Important points about the approach:

- The GPU can be kept busy, so long as the CPU is able to keep up with completed workloads.
- On the CPU, each workload is started in one frame and used in the subsequent frame.
- On the GPU, each workload is handled normally.
- To support this, there must be 2 sets of each buffer, and use of them alternates from one frame to the next. (This is the source of the term *double buffering*.) The `Cuper::DoubleBuffer` implementation keeps track of the correct buffer automatically.

An example usage of `Cuper::DoubleBuffer` might look like the following, with differences from `Cuper::Std` highlighted:

```
__global__ void Persistent (Cuper::DoubleBuffer::Token  token,
                            float* A0, float* A1)
{
      Cuper::DoubleBuffer::Cuda1Block  p(token);
      for (…) {
            p.waitForWork();
            unsigned int  which = p.claimBuffer();
            float*  A = which ? A1 : A0;
            …
            p.completeWork();
      }
}
```

```
void cpuFunction (…)
{
	Cuper::DoubleBuffer::Cpu  p;
	…
	Persistent<<<…>>>(p.token(), d_A0, d_1);

	// Start pipeline with an initial workload
	unsigned int  which = p.claimBuffer();
	float* h_A = which ? h_A1 : h_A0;
	… initialize h_A for workload 0…
	p.startCuda();

	for (unsigned int i = 1; …; i++) {
		which = p.nextBuffer();
		float* h_A = which ? h_A1 : h_A0;
		… initialize h_A for workload i…
		p.startCuda();

		… possibly do unrelated CPU work here …

		p.waitForCuda();
		which = p.claimBuffer();
		float*  h_A = which ? h_A1 : h_A0;
		… consume results in h_A for workload (i-1)…
	}

	// Flush last workload from pipeline
	p.waitForCuda();
	which = p.claimBuffer();
	float*  h_A = which ? h_A1 : h_A0;
	… consume results in h_A for workload i…

	p.terminateCuda();
}
```

Conclusion

The results presented in this paper demonstrate that the persistent threads programming model is a viable method for the usage of CUDA in hard real-time applications with tight sub-millisecond frame duration requirements. The CuPer API provides an easy way of accomplishing this. Integration of this approach with CPU code using the features of RedHawk Linux provides an application with strong determinism characteristics.

About Concurrent Real-Time

Concurrent Real-Time is the industry's foremost provider of high-performance real-time Linux® computer systems, solutions and software for commercial and government markets. The Company focuses on hardware-in-the-loop and man-in-the-loop simulation, data acquisition, and industrial systems, and serves industries that include aerospace and defense, automotive, energy and financial. Concurrent Real-Time is located in Pompano Beach, Florida with offices through North America, Europe and Asia.

Concurrent Real-Time's RedHawk Linux is an industry standard, real-time version of the open source Linux operating system for Intel x86 and ARM64 platforms. RedHawk Linux provides the guaranteed performance needed in time-critical and hard real-time environments. RedHawk is optimized for a broad range of server and embedded applications such as modeling, simulation, data acquisition, industrial control and medical imaging systems. RedHawk guarantees that a user-level application can respond to an external event in less than 5 microseconds on certified platforms.

For more information, please visit Concurrent Real-Time at www.concurrent-rt.com.