Performant Programming for GPUs

Thematic CERN School of Computing, Spring 2021

Daniel Hugo Cámpora Pérez
dcampora@cern.ch

17 Jun 2021

Table of Contents

Dealing with memory efficiently

Streams

Under the hood

Debugging and profiling

Summary

Resources

GPU performance

There are many considerations to be made when thinking of performance.

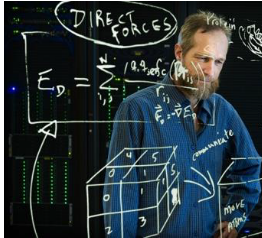


Figure: Computational Scientist[1]

[1] Source: US DoE

GPU performance

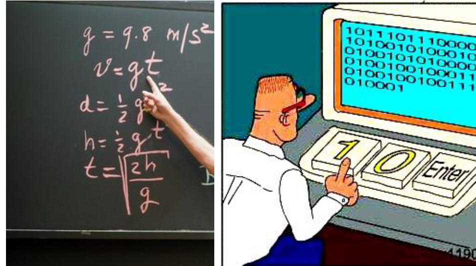


Figure: Domain Scientist – Performance Engineer

Table of Contents

Dealing with memory efficiently

Streams

Under the hood

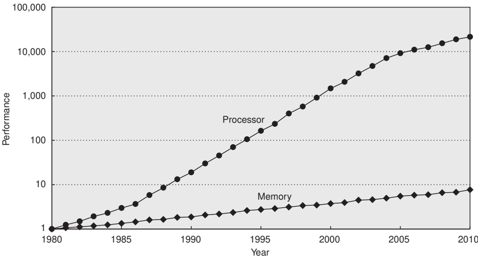
Debugging and profiling

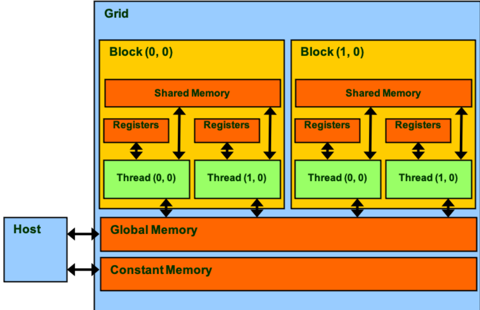
Summary

Resources

Price to pay for memory

As you already know, memory is a key item to consider when optimizing a program





Data locality

- Space locality – Neihbouring memory locations are likely going to be accessed.
- Temporal locality – The same memory location is likely going to be accessed again.

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7

Accessing x_0 will load into cache all x elements.

DRAM is read into cache memory – Each read brings a group of items onto cache memory.

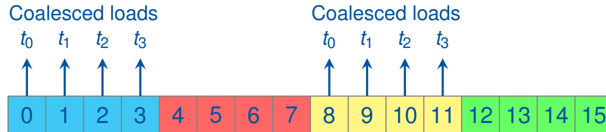
DRAM burst sections

In fact, DRAM is organized in **burst sections**. Let's take a simplified example:

- Each cell represents a byte
- We have a 16-byte address space, with 4-byte burst sections

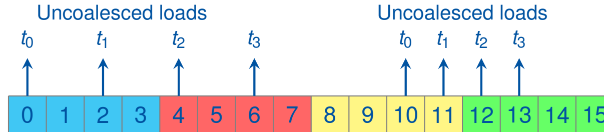
Note that nowadays the address spaces are in the GBs, and a typical burst section is 128 bytes.

Coalesced memory accesses



When threads make a memory request and the request falls under the same burst, the access is **coalesced**.

Coalesced memory accesses



However, if threads request a block of memory and the accesses do not fall under the same burst, the access is **uncoalesced**.

Several access patterns can yield this undesired behaviour, which impacts performance.

Is an access coalesced?

As a general rule, look for the following conditions:

- Base address should be a multiple of burst size.
- `threadIdx` should be used as a free term.

```
1  A[(expression independent of threadIdx) + threadIdx.x]
```

Listing: Coalesced accesses.

Quickbit: Linear representation of a matrix

$A_{0,0}$	$A_{1,0}$	$A_{2,0}$	$A_{3,0}$
$A_{0,1}$	$A_{1,1}$	$A_{2,1}$	$A_{3,1}$
$A_{0,2}$	$A_{1,2}$	$A_{2,2}$	$A_{3,2}$
$A_{0,3}$	$A_{1,3}$	$A_{2,3}$	$A_{3,3}$

is actually stored as

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

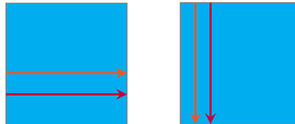
Tip: Always store higher order arrays as 1-dimensional arrays!

Matrix-matrix multiplication

Suppose we want to multiply two arrays:

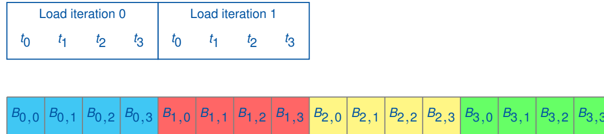
- **A** of size $m \times n$
- **B** of size $n \times k$
- Result is **C** of size $m \times k$

```
__device__ void multiply_arrays(float* A, float* B, float* C, int m, int n, int k) {
  for (int row = threadIdx.x; row < m; row += blockDim.x) {
    for (int col = 0; col < k; ++col) {
      float element = 0.f;
      for (int i = 0; i < n; ++i) {
        element += A[row * m + i] * B[i * k + col];
      }
      C[row * k + col] = element;
    }
  }
}
```



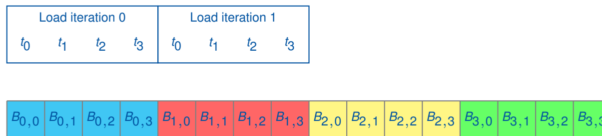
Access patterns

Accesses to B are **coalesced**:

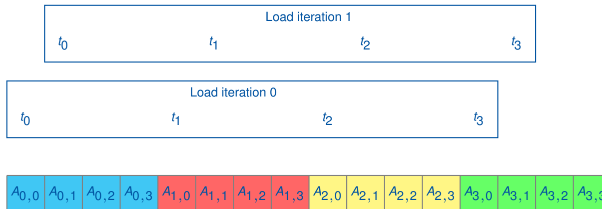


Access patterns

Accesses to B are **coalesced**:

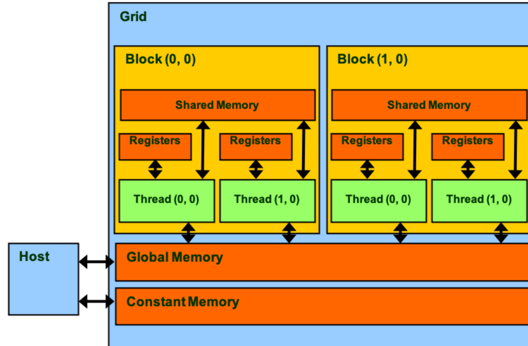


however, accesses to A are **uncoalesced**:



A step further: Shared memory

Going back to the types of memories available in a GPU:



Shared memory is a low-latency memory that resides on L1 cache.

How to use shared memory

Shared memory can be defined by using the keyword `__shared__`.

Any variable declared like this will be accessible by **all threads in a block**.

```
__global__ void shared_memory_example(float* dev_array) {
  __shared__ float array [256];

  for (int i = threadIdx.x; i < 256; i += blockDim.x) {
    array[i] = dev_array[i];
  }

  __syncthreads();

  // Now all threads can access array, which is initialized with
  // the first 256 elements of dev_array.
}
```

Listing: Coalesced accesses.

Things to consider about shared memory

Shared memory is a scarce resource that should be used carefully.

- It is limited in size, the maximum varies depending on the architecture.
- It is a limiting resource that is used to determine maximum number of blocks in flight in a Streaming Multiprocessor (SM).

The amount of memory reserved for L1-cache / shared memory is configurable[2].

At the same time, a good use of shared memory can lead to juicy performance gains!

[2] In CUDA, it can be configured with `cudaDeviceSetCacheConfig`

Matrix multiplication with shared memory

With a small enough matrix we could use shared memory:

- Preload all elements of **A** and **B** onto shared memory.
- Perform matrix multiplication reading from shared memory and store the result in **C**.

Bonus point: We can use **coalesced accesses** to populate the shared memory buffers!

Small matrix multiplication example

Load A and B onto shared memory:

Load iteration 0

	t_0	t_1	t_2	t_3												
A:	$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$
shared A:	$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$												
shared B:																

Small matrix multiplication example

Load A and B onto shared memory:

Load iteration 1

t_0 t_1 t_2 t_3

A:

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

shared A:

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$								

shared B:

Small matrix multiplication example

Load A and B onto shared memory:

Load iteration 2

t_0 t_1 t_2 t_3

A:

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

shared A:

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$				

shared B:

Small matrix multiplication example

Load A and B onto shared memory:

Load iteration 3

t_0 t_1 t_2 t_3

A:

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

shared A:

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

shared B:

Small matrix multiplication example

Load A and B onto shared memory:

Load iteration 4

t_0 t_1 t_2 t_3

B:

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$	$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$	$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$	$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

shared A:

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

shared B:

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$												

Small matrix multiplication example

Load A and B onto shared memory:

Load iteration 5

	t_0	t_1	t_2	t_3												
B:	$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$	$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$	$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$	$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$
shared A:	$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$
shared B:	$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$	$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$								

Small matrix multiplication example

Load A and B onto shared memory:

Load iteration 6

	t_0	t_1	t_2	t_3												
B:	$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$	$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$	$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$	$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$
shared A:	$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$
shared B:	$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$	$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$	$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$				

Small matrix multiplication example

Load A and B onto shared memory:

Load iteration 7

t_0 t_1 t_2 t_3

B:

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$	$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$	$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$	$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

shared A:

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

shared B:

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$	$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$	$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$	$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

Small matrix multiplication example (2)

shared A:

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

shared B:

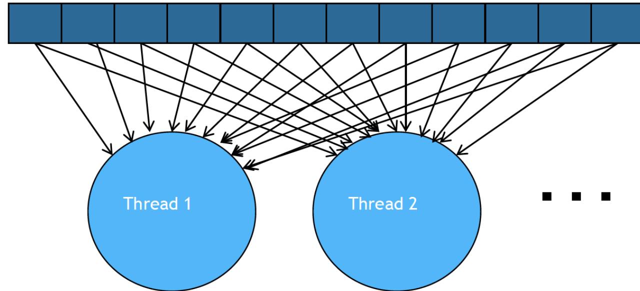
$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$	$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$	$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$	$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

And finally do the matrix-matrix multiplication from shared memory buffers **shared A** and **shared B**, storing it in **C**.

```
1  __global__ void shared_matrix_multiply_16_16(float* A, float* B, float* C) {
2    __shared__ float shared_A [256];
3    __shared__ float shared_B [256];
4
5    // Coalesced loads
6    for (int i = threadIdx.x; i < 256; i += blockDim.x)
7      shared_A[i] = A[i];
8    for (int i = threadIdx.x; i < 256; i += blockDim.x)
9      shared_B[i] = B[i];
10   __syncthreads();
11
12   // Now shared_A and shared_B are populated and can be used
13   // instead of the original arrays to perform the multiplication
14   multiply_arrays(shared_A, shared_B, C, 16, 16, 16);
15 }
```

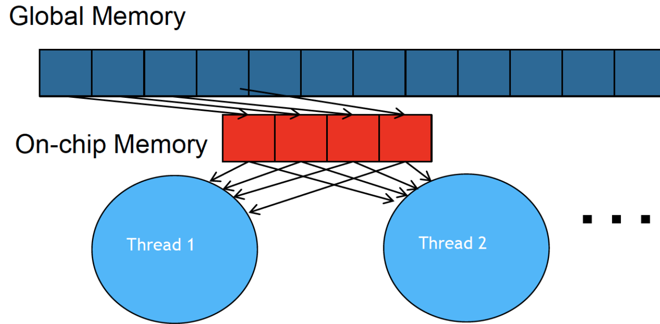
Tiling

Tiled data processing or just **tiling** consists in dividing a big chunk of data into many tiles which are processed one at a time.



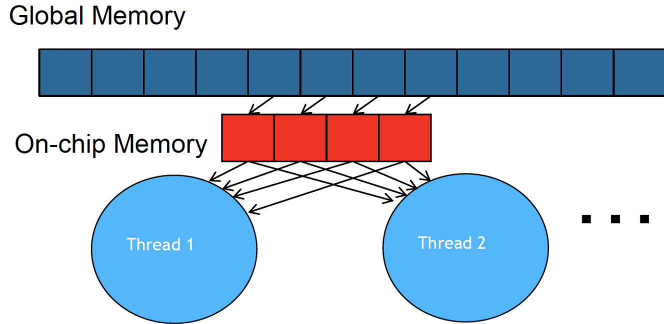
Tiling

Tiled data processing or just **tiling** consists in dividing a big chunk of data into many tiles which are processed one at a time.



Tiling

Tiled data processing or just **tiling** consists in dividing a big chunk of data into many tiles which are processed one at a time.



Analogy

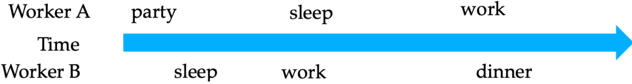
The basic concept is similar to carpooling:

- Drivers / Passengers – threads accessing memory
- Cars – memory access requests

Schedule is important!

It works well when people have similar schedules

But it goes really wrong otherwise!



A generic tiling algorithm

Follow the next steps:

- Identify an access pattern where threads access global memory in a tiled manner
- Load the tile from global into shared memory in a coalesced manner
- Synchronize
- Have multiple threads access the data from the shared buffer
- Synchronize
- Move on to the next tile

Tiled matrix multiplication

Using this technique we can multiply two arrays *of any given size* by dividing it into tiles.

At every step, we will load the data into shared memory and perform the multiplication.

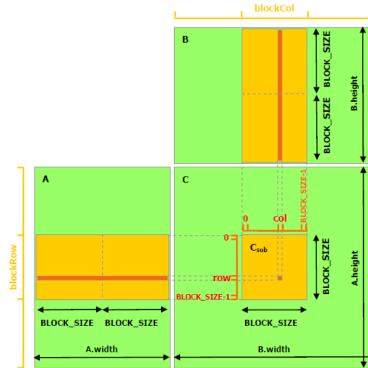


Table of Contents

Dealing with memory efficiently

Streams

Under the hood

Debugging and profiling

Summary

Resources

Streams

A **Stream** is a sequence of commands that execute in order.

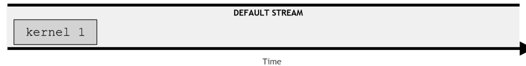
A Stream can execute various types of commands. For instance,

- Kernel invocations
- Memory transmissions
- Memory (de)allocations
- Memsets
- Synchronizations

The default stream

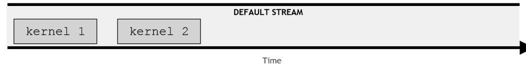
CUDA has a default Stream

The default stream (2)



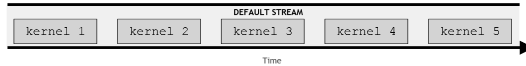
By default, CUDA kernels run in the default stream

The default stream (3)



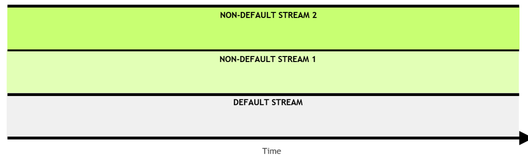
Any instruction run in a stream must complete before the next can be issued

The default stream (3)



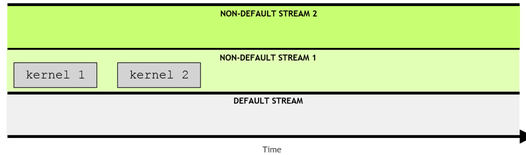
Any instruction run in a stream must complete before the next can be issued

Non-default streams



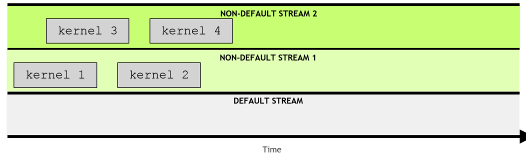
Non-default streams can also be created in a CUDA application

Non-default streams (2)



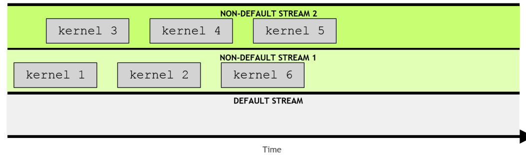
Commands running on a non-default stream must still complete before the next can be issued

Non-default streams (3)



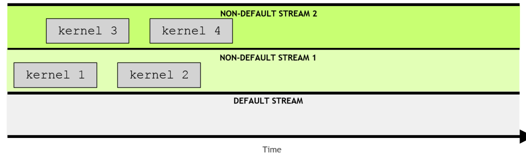
However, commands in different, non-default streams can run concurrently

Non-default streams (3)



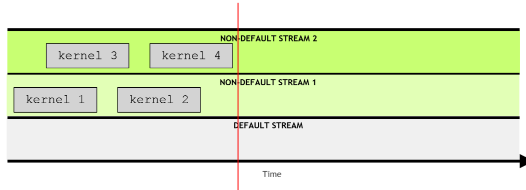
However, commands in different, non-default streams can run concurrently

The default stream is blocking



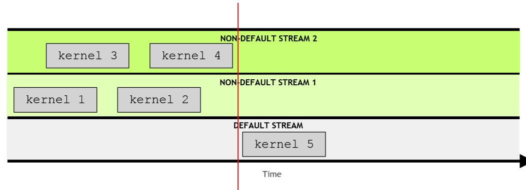
The default stream is special: it acquires exclusive access preventing other streams from running

The default stream is blocking



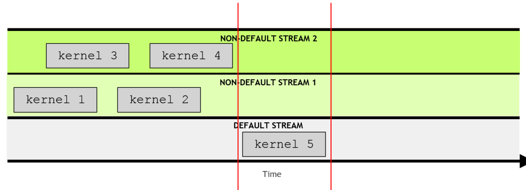
The default stream is special: it acquires exclusive access preventing other streams from running

The default stream is blocking



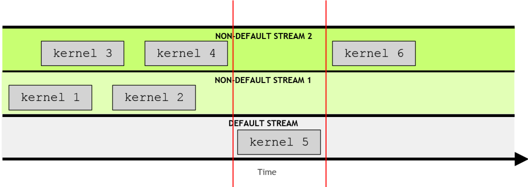
The default stream is special: it acquires exclusive access preventing other streams from running

The default stream is blocking



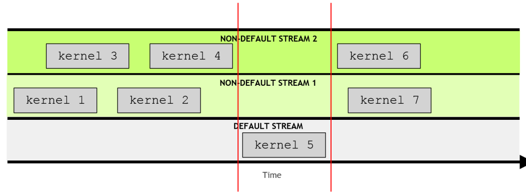
The default stream is special: it acquires exclusive access preventing other streams from running

The default stream is blocking



The default stream is special: it acquires exclusive access preventing other streams from running

The default stream is blocking



The default stream is special: it acquires exclusive access preventing other streams from running

Pipelines

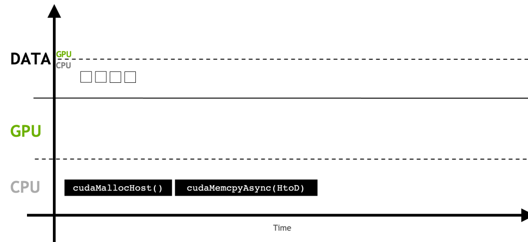
If we were to use a single stream to perform all calculations and transfer all data, GPUs would be *hopelessly slow*.

Thankfully, GPUs can perform data transmissions while executing kernels.

Given that a GPU is sitting on a PCI-express slot, we can even exploit the full-duplex capability of the link if we so desire. Typically at least three streams are needed to achieve a full pipeline:

- Use SMs to perform some computation
- Transfer data host-to-device
- Transfer data device-to-host

Pipeline example



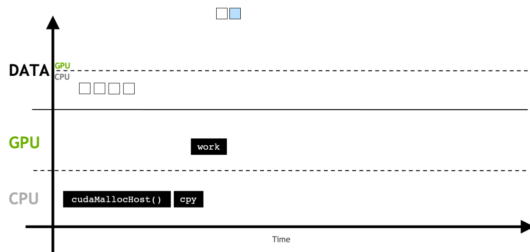
Main memory (host) must be **pinned** in order for asynchronicity to work

`cudaMemcpyAsync` can transfer data **asynchronously** in a **non-default stream**

Pipeline example (2)

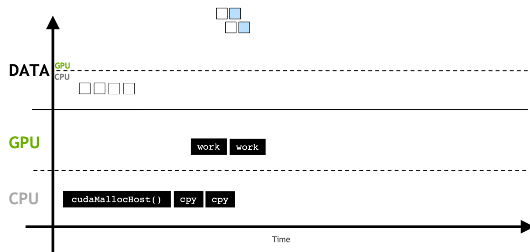
This allows overlapping memory copies and computation

Pipeline example (2)



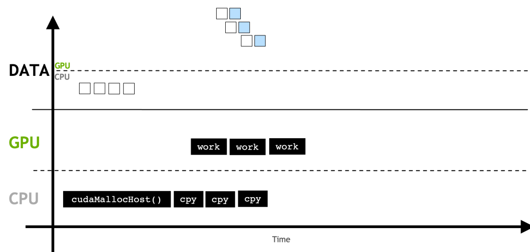
This allows overlapping memory copies and computation

Pipeline example (2)



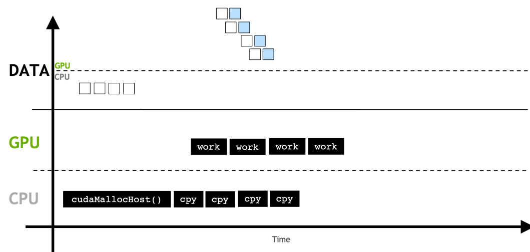
This allows overlapping memory copies and computation

Pipeline example (2)



This allows overlapping memory copies and computation

Pipeline example (2)



This allows overlapping memory copies and computation

Full pipeline

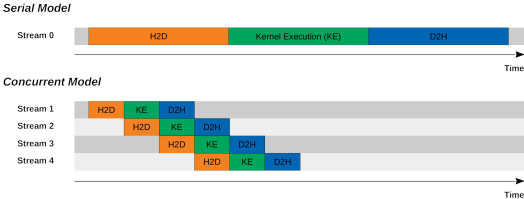


Figure: Source

A fully realized pipeline.

Table of Contents

Dealing with memory efficiently

Streams

Under the hood

Debugging and profiling

Summary

Resources

Streaming Multiprocessor

The processor that performs computations in NVIDIA architectures is the **Streaming Multiprocessor**. It consists of

- Arithmetic (green)
- Load / store (red)
- Memory (blue)
- Control unit (orange)

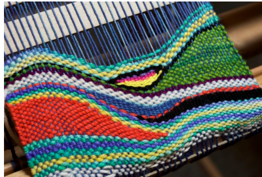
There are many SMs on a GPU, current models have up to 80 SMs and thousands of "CUDA cores"

The warp

The SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called **warps**

Threads inside a warp:

- start at the same program address
- have their own program counter (instruction address counter)
- have their own register state



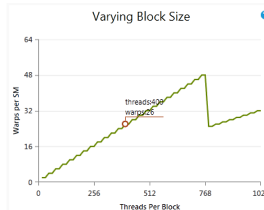
Does warp size matter?

Warp size affects what block size configurations are efficient. It affects *occupancy*:

Occupancy is the ratio of active warps to maximum supported active warps in a SM.

Example: On a GPU that supports **64** active warps per SM, full occupancy on a SM can be achieved with:

- 8 active blocks with 256 threads per block.
- 16 active blocks with 128 threads per block.

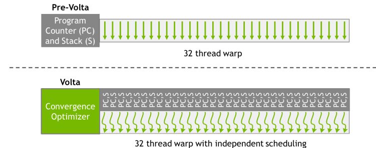


Lockstep

Older architectures executed in *lockstep*, they shared program counter and register state.

However, this assumption is not valid anymore.

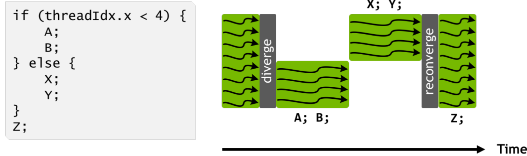
Threads can now branch and execute independently.



What happens with branches?

If you are working on a GPU that runs in lockstep, if there is at least one thread running the branch then the whole warp will go through the branch.

For this reason, it is commonly said that one should avoid branches when writing GPU code:

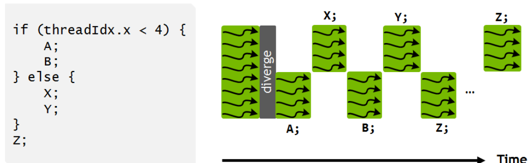


Branchless code

As compilers get smarter and GPUs do not execute in lockstep, it is not the case anymore that one should avoid branches at any cost. *In most cases, branches are ok.*

In recent models, threads within a warp are **scheduled independently**:

- Execution of statements can be interleaved.
- At one clock cycle, one single same instruction is executed for all threads in a warp (SIMT).



Are branches relevant?

When branches lead to **homogeneous code** then it is worth removing the branch. Especially if the code behind the branch is a *hot section* and *complex* for the compiler. In essence:

Avoid long sequences of diverged execution by threads within the same warp.

For instance, given a seed of a particle trajectory, find compatible hits in other sensors:

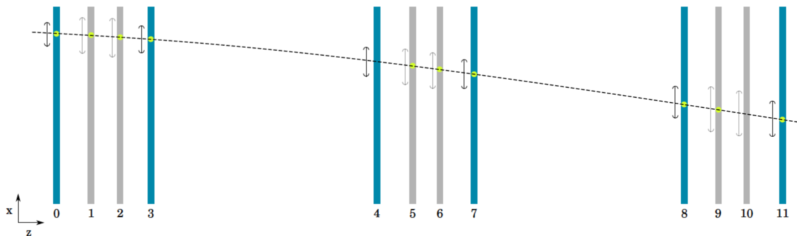


Table of Contents

Dealing with memory efficiently

Streams

Under the hood

Debugging and profiling

Summary

Resources

Debugging

GPU code can be debugged in a similar way to how CPU code is debugged.

There are several tools that can be used for debugging GPU program's execution:

- `cuda-gdb` – Command line debugger that is based off the popular `gdb`. It can be used to debug CUDA applications, set watchpoints, step into execution of any thread and so on.
- **NVIDIA Nsight** – Nsight is both an extension to Visual Studio and an extension to the Eclipse environment that adds CUDA support. The Visual Studio version is the better of the two, and it contains a built-in debugger and profiler. It is fully integrated with the IDE, so breakpoints can be set, values can be expanded, just like with the CPU debugger.
- `rocgdb` – Command line debugger that supports the ROCm tool suite. It is at the prototype stage but it is a huge improvement over the previous debugging capabilities of ROCm.

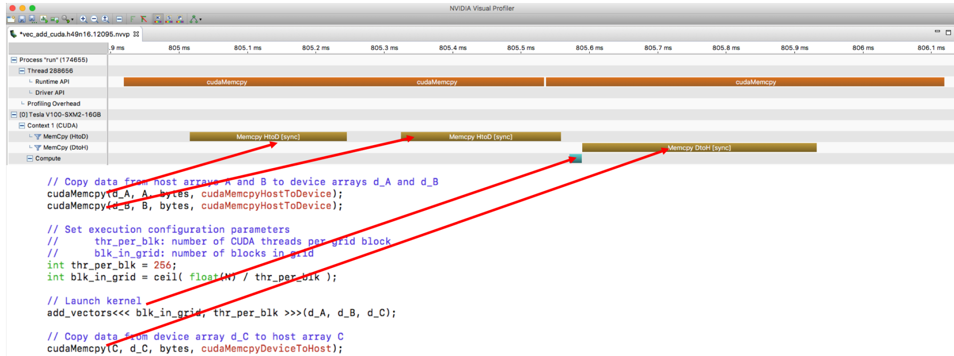
Profiling

Similarly, for profiling there are various tools to check out:

- `nsys` – Command line profiler that replaces the previous `nvprof`. It is highly configurable and can produce analytics that can be analyzed with the visual profiler.
- **NVIDIA Visual Profiler (nvvp), Nsight Systems, Nsight Compute** – These three tools provide complementary analytics and functionalities to optimize your application. It is also possible to connect remotely to a server where the application is run, results are collected and presented in the local profiler instance.
- `rocprof` – The command line profiler of ROCm is in its infancy but provides decent functionality. It supports generating traces that can be opened with 3rd party tools and visualized eg. in a browser.

What to look for when profiling

As a general rule, profile your code often and keep track of optimizations you performed.



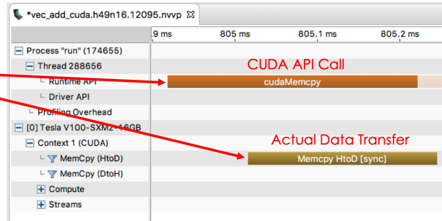
What to look for when profiling (2)

```
// Copy data from host arrays A and B to device arrays d_A and d_B
cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

// Set execution configuration parameters
//      thr_per_blk: number of CUDA threads per grid block
//      blk_in_grid: number of blocks in grid
int thr_per_blk = 256;
int blk_in_grid = ceil( float(N) / thr_per_blk );

// Launch kernel
add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);

// Copy data from device array d_C to host array C
cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
```



Details about the data transfer

Properties

Memcpy HtoD [sync]

Start	805.016 ms (805,015,657...
End	805.245 ms (805,245,320...
Duration	229.663 µs
Size	4.194 MB
Throughput	18.263 GB/s
Stream	Default
Memory Type	
Source	Pageable
Destination	Device

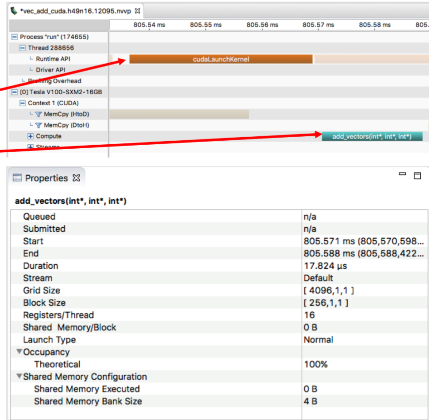
What to look for when profiling (3)

```
// Copy data from host arrays A and B to device arrays d_A and d_B
cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

// Set execution configuration parameters
//      thr_per_blk: number of CUDA threads per grid block
//      blk_in_grid: number of blocks in grid
int thr_per_blk = 256;
int blk_in_grid = ceil( float(N) / thr_per_blk );

// Launch kernel
add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);

// Copy data from device array d_C to host array C
cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
```

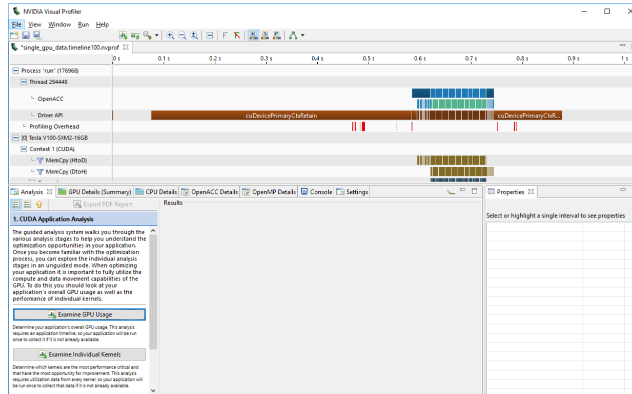


Queued	n/a
Submitted	n/a
Start	805.571 ms (805,570,598...
End	805.588 ms (805,588,422...
Duration	17.824 µs
Stream	Default
Grid Size	[4096,1,1]
Block Size	[256,1,1]
Registers/Thread	16
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	100%
▼Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

Details about the kernel execution

Perform a guided analysis

`nvvp` provides a guided analysis that takes a top-bottom approach, zooming into problems and not overwhelming with too much information.



Perform a guided analysis (1)

Perform a guided analysis (2)

NVIDIA Visual Profiler

File View Window Run Help

*single_gpu_data.timeline100.nvprof | *vec_add_cuda.timeline.nvprof

0 s 0.05 s 0.1 s 0.15 s 0.2 s 0.25 s 0.3 s 0.35 s 0.4 s 0.45 s 0.5 s

- Process "run" (2129)
  - Thread 288400
    - Runtime API
    - Driver API
  - Profiling Overhead
- [0] Tesla V100-SXM2-16GB
  - Context 1 (CUDA)
    - MemCpy (HtoD)
    - MemCpy (DtoH)

Analysis | GPU Details (Summary) | CPU Details | OpenACC Details | OpenMP Details | Console | Settings

Export PDF Report

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound
4. Instruction and Memory Latency

Instruction and memory latency limit the performance of a kernel when the GPU does not have enough work to keep busy. The performance of latency-limited kernels can often be improved by increasing occupancy. Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. Theoretical occupancy provides an upper bound while achieved occupancy indicates the kernel's actual occupancy.

Examine Occupancy

Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. Theoretical occupancy provides an upper bound while achieved occupancy indicates the kernel's actual occupancy.

Show Kernel Profile - PC Sampling

The kernel profile shows the samples of various stall reasons collected at each pc of the assembly instruction. Using this information you can pinpoint portions

Results

Occupancy Is Not Limiting Kernel Performance

The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU. More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [4096,1,1] (4096 blocks)Block Size: [256,1
Occupancy Per SM				
Active Blocks		8	32	0 4 8 12 16 20 24 28 32
Active Warps	53.8	64	64	0 9 18 27 36 45 54 64
Active Threads		2048	2048	0 512 1024 1536 2048
Occupancy	84.1%	100%	100%	0% 25% 50% 75% 100%
Warps				
Threads/Block		256	1024	0 256 512 768 1024
Warps/Block		8	32	0 4 8 12 16 20 24 28 32
Block Limit		8	32	0 4 8 12 16 20 24 28 32
Registers				
Registers/Thread		16	65536	

Prop...

add_vectors(int*, int*,...

- Queued
- Submitted
- Start
- End
- Duration
- Stream
- Grid Size
- Block Size
- Registers/Thread
- Shared Memory/Blc
- Launch Type
- Occupancy
  - Achieved
  - Theoretical
- Shared Memory Cor
  - Shared Memory
  - Shared Memory

Table of Contents

Dealing with memory efficiently

Streams

Under the hood

Debugging and profiling

Summary

Resources

Summary

- Prefer coalesced memory accesses.
- Shared memory is a valuable resource, use it, don't abuse it.
- Tiling is a technique that helps optimize memory performance.

- Use streams to optimize GPU usage.
- Pipelines with three or more streams yield best results.

- Warp size is a hardware detail that affects efficient block sizes.
- Avoid branches but don't go paranoid.
- Profile, profile, profile.

We will see some of these concepts in the exercises this afternoon.

Table of Contents

Dealing with memory efficiently

Streams

Under the hood

Debugging and profiling

Summary

Resources

Resources used in the talk

- GPU Teaching Kit on Accelerated Computing
- NVIDIA Deep Learning Institute materials
- Talk on NVIDIA Profiling Tools by Jeff Larkin