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 – Neighbouring memory locations are likely going to be accessed.
▶ Temporal locality – The same memory location is likely going to be accessed again.

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.
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.
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.

Listing: Coalesced accesses.
Quickbit: Linear representation of a matrix

\[
\begin{array}{cccc}
A_{0,0} & A_{1,0} & A_{2,0} & A_{3,0} \\
A_{0,1} & A_{1,1} & A_{2,1} & A_{3,1} \\
A_{0,2} & A_{1,2} & A_{2,2} & A_{3,2} \\
A_{0,3} & A_{1,3} & A_{2,3} & A_{3,3} \\
\end{array}
\]

is actually stored as

\[
\begin{array}{cccccccccccc}
A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} & A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} & A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} & A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \\
\end{array}
\]

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 $m \times k$

```c
__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:

<table>
<thead>
<tr>
<th>Load iteration 0</th>
<th>Load iteration 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$  $t_1$  $t_2$  $t_3$</td>
<td>$t_0$  $t_1$  $t_2$  $t_3$</td>
</tr>
</tbody>
</table>

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

However, accesses to A are uncoalesced:
Access patterns

Accesses to B are coalesced:

<table>
<thead>
<tr>
<th>Load iteration 0</th>
<th>Load iteration 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$  $t_1$  $t_2$  $t_3$</td>
<td>$t_0$  $t_1$  $t_2$  $t_3$</td>
</tr>
</tbody>
</table>

| 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$_{3,3}$ |

however, accesses to A are uncoalesced:

<table>
<thead>
<tr>
<th>Load iteration 0</th>
<th>Load iteration 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$  $t_1$  $t_2$  $t_3$</td>
<td>$t_0$  $t_1$  $t_2$  $t_3$</td>
</tr>
</tbody>
</table>

| 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$_{3,3}$ |
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:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$</td>
<td>$t_1$</td>
<td>$t_2$</td>
<td>$t_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{0,0}$</td>
<td>$A_{0,1}$</td>
<td>$A_{0,2}$</td>
<td>$A_{0,3}$</td>
<td>$A_{1,0}$</td>
<td>$A_{1,1}$</td>
<td>$A_{1,2}$</td>
<td>$A_{1,3}$</td>
</tr>
</tbody>
</table>

shared A:

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

shared B:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Small matrix multiplication example

Load A and B onto shared memory:

<table>
<thead>
<tr>
<th></th>
<th>A0,0</th>
<th>A0,1</th>
<th>A0,2</th>
<th>A0,3</th>
<th>A1,0</th>
<th>A1,1</th>
<th>A1,2</th>
<th>A1,3</th>
<th>A2,0</th>
<th>A2,1</th>
<th>A2,2</th>
<th>A2,3</th>
<th>A3,0</th>
<th>A3,1</th>
<th>A3,2</th>
<th>A3,3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>shared A:</strong></td>
<td>A0,0</td>
<td>A0,1</td>
<td>A0,2</td>
<td>A0,3</td>
<td>A1,0</td>
<td>A1,1</td>
<td>A1,2</td>
<td>A1,3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>shared B:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Small matrix multiplication example

Load A and B onto shared memory:

\[
\text{A:} \quad \begin{array}{cccccccc}
A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} & A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\
A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} & A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3}
\end{array}
\]

\[
\text{shared A:} \quad \begin{array}{cccccccc}
A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} & A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\
A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} & \hline & \hline
\end{array}
\]

\[
\text{shared B:} \quad \begin{array}{cccccccc}
 & & & & & & & \\
 & & & & & & & \\
\end{array}
\]
Small matrix multiplication example

Load A and B onto shared memory:

<table>
<thead>
<tr>
<th>A:</th>
<th>A₀,₀</th>
<th>A₀,₁</th>
<th>A₀,₂</th>
<th>A₀,₃</th>
<th>A₁,₀</th>
<th>A₁,₁</th>
<th>A₁,₂</th>
<th>A₁,₃</th>
<th>A₂,₀</th>
<th>A₂,₁</th>
<th>A₂,₂</th>
<th>A₂,₃</th>
<th>A₃,₀</th>
<th>A₃,₁</th>
<th>A₃,₂</th>
<th>A₃,₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared A:</td>
<td>A₀,₀</td>
<td>A₀,₁</td>
<td>A₀,₂</td>
<td>A₀,₃</td>
<td>A₁,₀</td>
<td>A₁,₁</td>
<td>A₁,₂</td>
<td>A₁,₃</td>
<td>A₂,₀</td>
<td>A₂,₁</td>
<td>A₂,₂</td>
<td>A₂,₃</td>
<td>A₃,₀</td>
<td>A₃,₁</td>
<td>A₃,₂</td>
<td>A₃,₃</td>
</tr>
<tr>
<td>shared B:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Load iteration 3

| t₀ | t₁ | t₂ | t₃ |
Small matrix multiplication example

Load A and B onto shared memory:

Load iteration 4

\[
\begin{array}{cccc}
& t_0 & t_1 & t_2 & t_3 \\
B: & B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} & \ldots & B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} & B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} & B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \\
\end{array}
\]

shared A:

\[
\begin{array}{cccc}
& A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} & A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} & A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} & A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \\
\end{array}
\]

shared B:

\[
\begin{array}{cccc}
& B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\end{array}
\]
Small matrix multiplication example

Load A and B onto shared memory:

<table>
<thead>
<tr>
<th>B:</th>
<th>A:</th>
<th>shared A:</th>
<th>shared B:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$</td>
<td>$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$</td>
<td>$A_{0,0}$ $B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$ $B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$</td>
<td></td>
</tr>
</tbody>
</table>
Small matrix multiplication example

Load A and B onto shared memory:

<table>
<thead>
<tr>
<th>B:</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_{0,0} B_{0,1} B_{0,2} B_{0,3} B_{1,0} B_{1,1} B_{1,2} B_{1,3}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t_0</th>
<th>t_1</th>
<th>t_2</th>
<th>t_3</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>shared A:</th>
</tr>
</thead>
<tbody>
<tr>
<td>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}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>shared B:</th>
</tr>
</thead>
<tbody>
<tr>
<td>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}</td>
</tr>
</tbody>
</table>

Daniel Cámpora
Small matrix multiplication example

Load A and B onto shared memory:

Load iteration 7

<table>
<thead>
<tr>
<th>t₀</th>
<th>t₁</th>
<th>t₂</th>
<th>t₃</th>
</tr>
</thead>
</table>

**B:**

<table>
<thead>
<tr>
<th>B₀₀</th>
<th>B₀₂</th>
<th>B₀₃</th>
<th>B₁₀</th>
<th>B₁₁</th>
<th>B₁₂</th>
<th>B₂₀</th>
<th>B₂₁</th>
<th>B₂₂</th>
<th>B₂₃</th>
<th>B₃₀</th>
<th>B₃₁</th>
<th>B₃₂</th>
<th>B₃₃</th>
</tr>
</thead>
</table>

**shared A:**

<table>
<thead>
<tr>
<th>A₀₀</th>
<th>A₀₂</th>
<th>A₀₃</th>
<th>A₁₀</th>
<th>A₁₁</th>
<th>A₁₂</th>
<th>A₂₀</th>
<th>A₂₁</th>
<th>A₂₂</th>
<th>A₂₃</th>
<th>A₃₀</th>
<th>A₃₁</th>
<th>A₃₂</th>
<th>A₃₃</th>
</tr>
</thead>
</table>

**shared B:**

| B₀₀ | B₀₂ | B₀₃ | B₁₀ | B₁₁ | B₁₂ | B₁₃ | B₂₀ | B₂₁ | B₂₂ | B₂₃ | B₃₀ | B₃₁ | B₃₂ | B₃₃ |
And finally do the matrix-matrix multiplication from shared memory buffers `shared A` and `shared B`, storing it in `C`.

```c
__global__ void shared_matrix_multiply_16_16(float* A, float* B, float* C) {
    __shared__ float shared_A[256];
    __shared__ float shared_B[256];

    // Coalesced loads
    for (int i = threadIdx.x; i < 256; i += blockDim.x)
        shared_A[i] = A[i];
    for (int i = threadIdx.x; i < 256; i += blockDim.x)
        shared_B[i] = B[i];
    __syncthreads();

    // Now shared_A and shared_B are populated and can be used
    // instead of the original arrays to perform the multiplication
    multiply_arrays(shared_A, shared_B, C, 16, 16, 16);
}
```
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.

Global Memory

Thread 1

Thread 2
**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.
Tiled data processing or just tiling consists in dividing a big chunk of data into many tiles which are processed one at a time.
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.
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
CUDA has a default Stream
The default stream (2)

By default, CUDA kernels run in the default stream
Any instruction run in a stream must complete before the next can be issued
Any instruction run in a stream must complete before the next can be issued
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
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
Main memory (host) must be **pinned** in order for asynchronicity to work

cudaMemcpyAsync can transfer data **asynchronously** in a **non-default stream**
This allows overlapping memory copies and computation
This allows overlapping memory copies and computation
This allows overlapping memory copies and computation
This allows overlapping memory copies and computation
This allows overlapping memory copies and computation
A fully realized pipeline.
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.
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.

![Diagram showing pre-Volta and Volta execution with and without independent scheduling]
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:

```c
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```
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
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.
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)

```c
// 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);
```
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);

Details about the kernel execution
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)

The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU’s available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to increase GPU utilization.

1. CUDA Application Analysis
2. Check Overall GPU Usage

You can also examine the performance of individual kernels to expose additional optimization opportunities.
Perform a guided analysis (2)
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