CME 213

SPRING 2019

Eric Darve

Stanford University

Review

"Secret" behind GPU performance: simple cores but a large number of them; even more threads can exist live on the hardware (10k–20k threads live).

Important performance bottleneck: data traffic between memory and register files

Programming on a GPU:

- Warp = 32 threads
- Block = group of threads running on one SM
- SM runs an integer number of thread blocks
- Grid = set of blocks representing the entire data set

Code example

- Let’s start with a very basic example
  `firstProgram.cu`

- Log onto GCP!

```
$ gcloud compute ssh hw3
$ gcloud compute scp * hw3:~/.
```

Compile with:

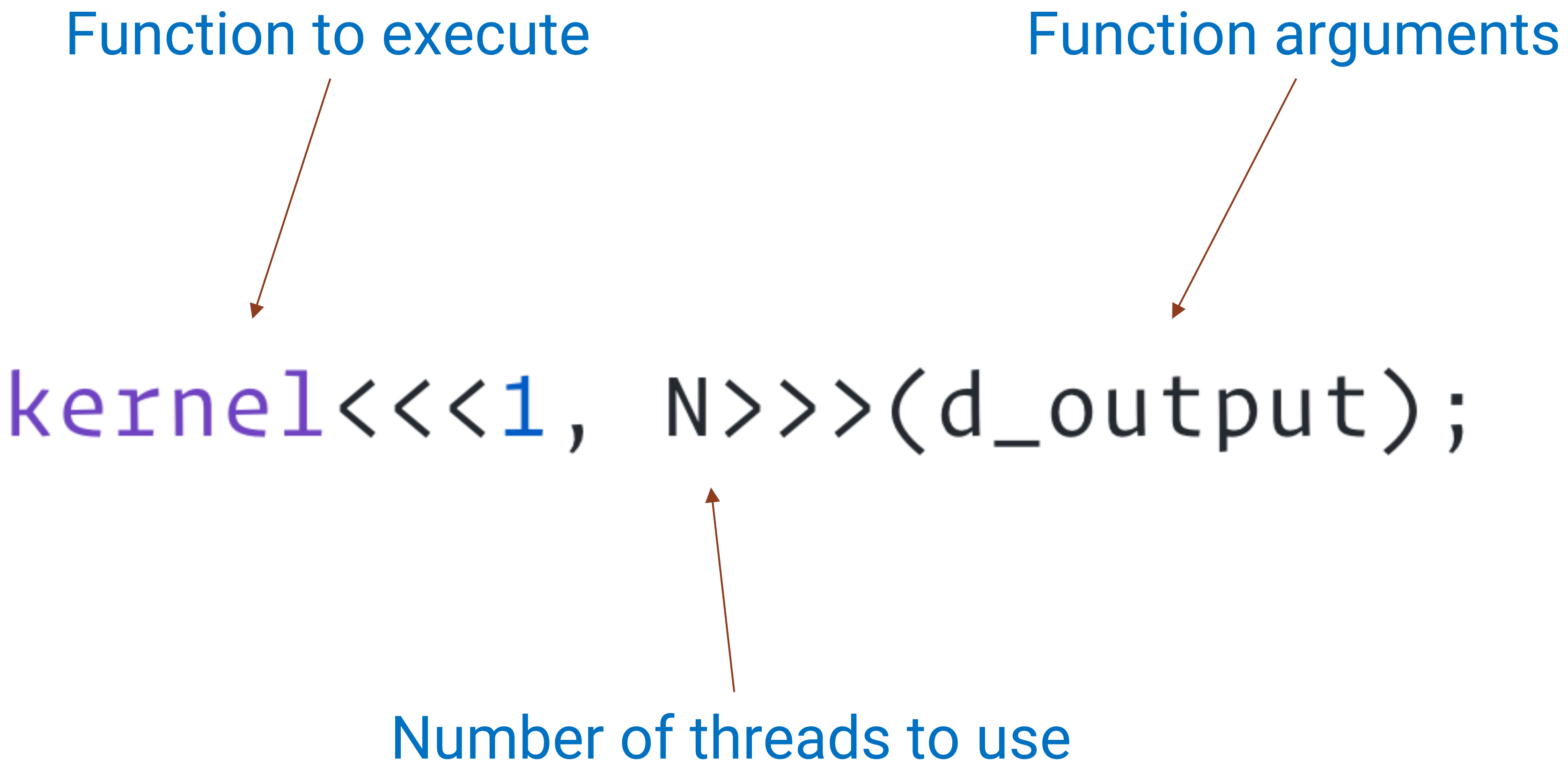
```
$ make
```

or:

```
$ nvcc --compiler-options -Wall -arch=sm_35 -o test test.cu
```

Code overview

```
int* d_output;
cudaMalloc(&d_output, sizeof(int) * N);
kernel<<<1, N>>>(d_output);
vector<int> h_output(N);
cudaMemcpy(&h_output[0], d_output, sizeof(int) * N, cudaMemcpyDeviceToHost);
for(int i = 0; i < N; ++i)
    printf("Entry %3d, written by thread %2d\n", h_output[i], i);
cudaFree(d_output);
```



```
kernel<<<1, N>>>(d_output);
```

Device kernels

```
__device__ __host__
int f(int i) {
    return i*i;
}

__global__
void kernel(int* out) {
    out[threadIdx.x] = f(threadIdx.x);
}
```

global host device

What are these mysterious keywords?

`__global__` kernel will be

- Executed on the device
- Callable from the host

`__host__` kernel will be ← Compiled for host

- Executed on the host
- Callable from the host

`__device__` kernel will be ← Compiled for device

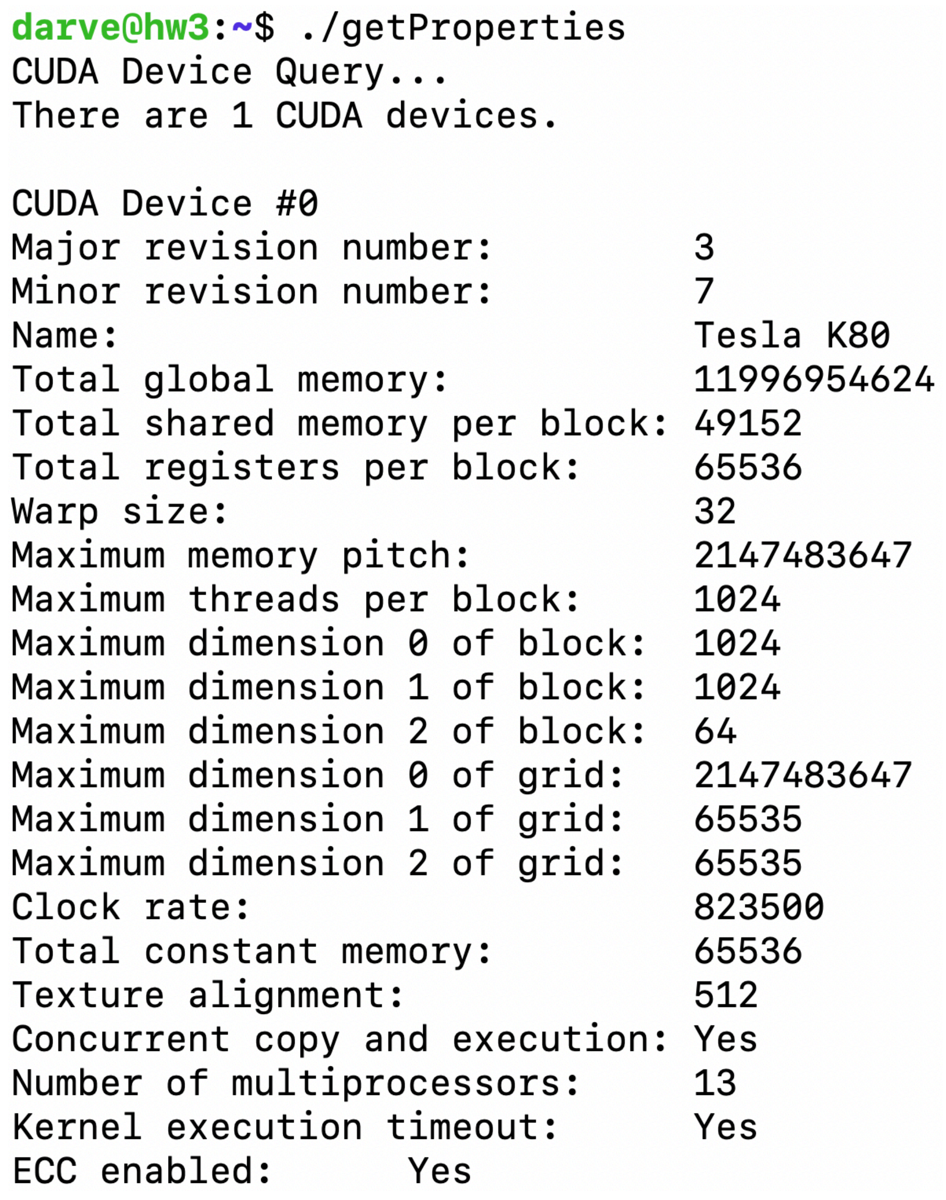
- Executed on the device
- Callable from the device only

Run the following!

```
./firstProgram 1
./firstProgram 32
./firstProgram 1024
./firstProgram 1025
```

Try with arguments:

1, 2, 3, 4, 16, 32, 64, 1024, **1025**



```
darve@hw3:~$ ./getProperties
CUDA Device Query...
There are 1 CUDA devices.

CUDA Device #0
Major revision number:         3
Minor revision number:         7
Name:                          Tesla K80
Total global memory:           11996954624
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    823500
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     13
Kernel execution timeout:      Yes
ECC enabled:             Yes
```

Thread hierarchy

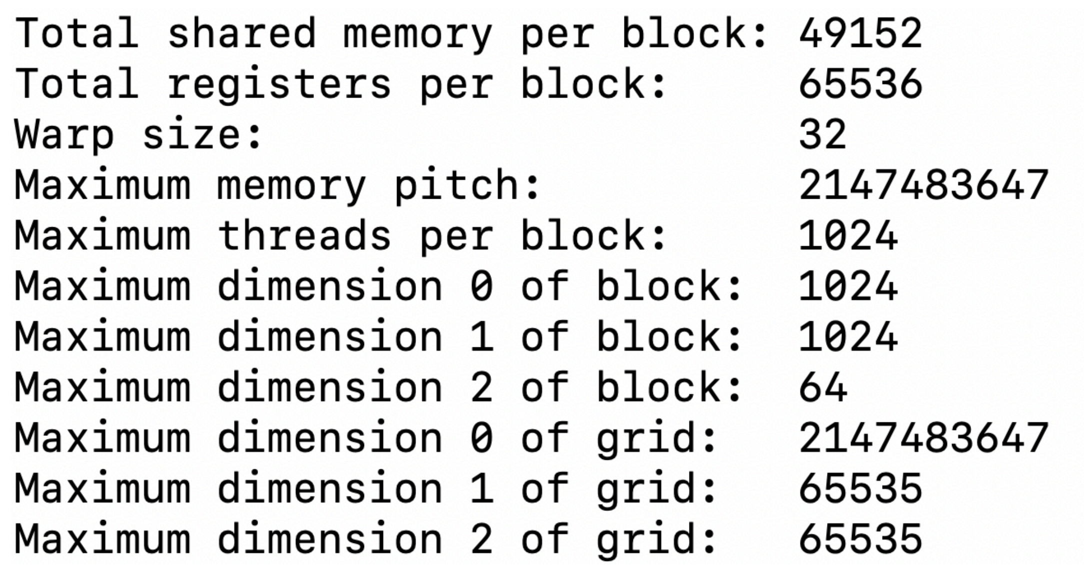
Two levels of hierarchy:

- **Thread block:** a block is loaded entirely on an SM. This means that a block of threads cannot require more resources than available, for example to store register variables.
- **Grid of blocks:** for large problems, we can use multiple blocks. A very large number of blocks can be allocated so that we can execute a kernel over a very large data set.

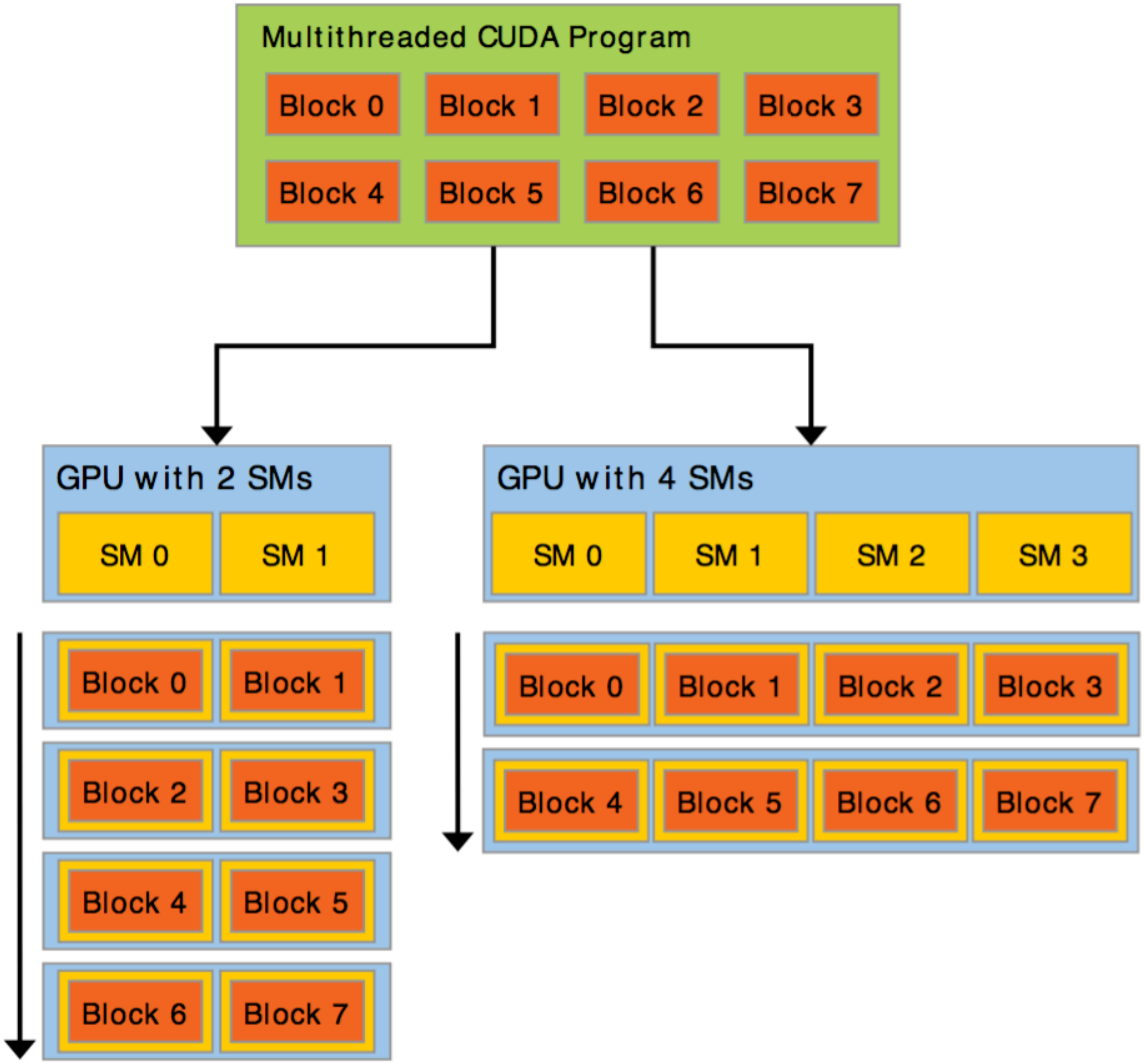
Problem decomposition

Decompose your problem into:

- Small blocks so that data can fit in SM.
- Typically the number of threads per block is around 256−512.
- Create a grid of blocks so that you can process the entire data.



```
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
```


Multithreaded CUDA Program
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 2 SMs
SM 0
SM 1
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 4 SMs
SM 0
SM 1
SM 2
SM 3
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7

Block execution

- The device will start by loading data on each SM to execute the blocks.
- At least one block must fit on an SM, e.g., there should be enough memory for register variables.
- The SM will load up as many blocks as possible until it runs out of resources.
- Then it will start executing the blocks, that is all threads will execute the kernel.
- Once all threads in a block are done with the kernel, another block gets loaded in the SM.
- And so on until the device runs out of blocks to execute.

Dimensions

Blocks and grids can be 1D, 2D or 3D.
Their dimensions is declared using:

```
dim3 threadsPerBlock(Nx);
dim3 numBlocks(Mx);

dim3 threadsPerBlock(Nx, Ny);
dim3 numBlocks(Mx, My);

dim3 threadsPerBlock(Nx, Ny, Nz);
dim3 numBlocks(Mx, My, Mz);
```

Block size

Thread index example:

```
int col = blockIdx.x * blockDim.x + threadIdx.x;
int row = blockIdx.y * blockDim.y + threadIdx.y;
```

Thread ID in block

Block ID

addMatrices.cu

```
__global__
void Add(int n, int* a, int* b, int* c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if(i < n && j < n) {
        c[n*i + j] = a[n*i + j] + b[n*i + j];
    }
}
```

```
dim3 threads_per_block(2,n_thread);
int blocks_per_grid_x = (n + 2 - 1) / 2;
int blocks_per_grid_y = (n + n_thread - 1) / n_thread;
dim3 num_blocks(blocks_per_grid_x, blocks_per_grid_y);
Initialize<<<num_blocks, threads_per_block>>>(n, d_a, d_b);
Add<<<num_blocks, threads_per_block>>>(n, d_a, d_b, d_c);
```

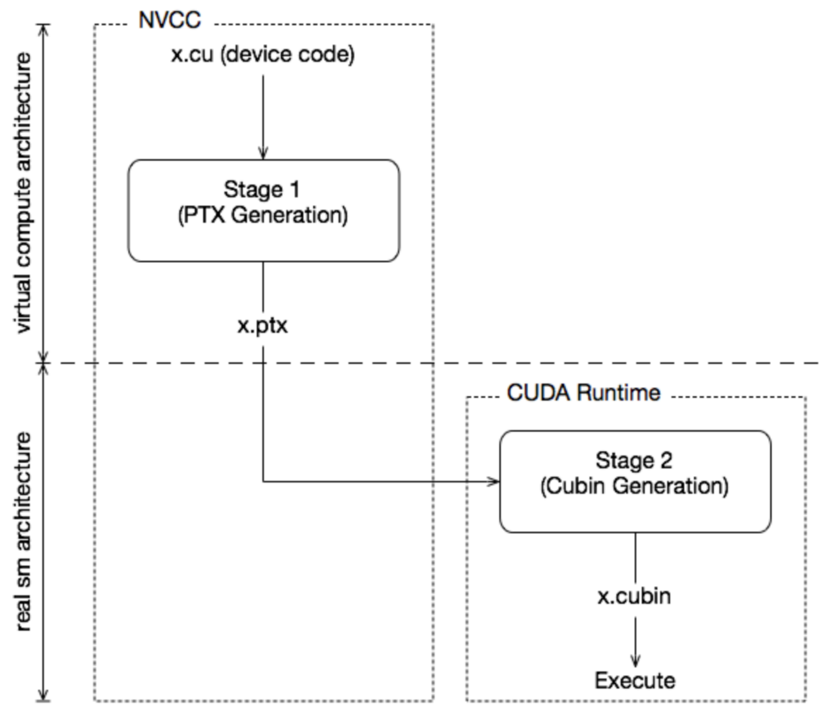
Compiler options and other tricks

`-g`	Debug on the host
`-G`	Debug on the device (CUDA-gdb, Nsight Eclipse Edition)
`-pg`	Profiling info for use with gprof (Linux only)
`-Xcompiler`	Options for underlying gcc compiler
`-O`	Optimization level

More on profiling: **`nvprof`**

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

GPU architecture



nvcc produces different types of codes:

- **Architecture:** a virtual architecture with certain capabilities. Only declare the capabilities your code is using (lowest possible number)
- **Real target GPU:** compilation of the code for a specific GPU (many can be specified)

The architecture and GPU code types can be specified:

```
$ nvcc code.cu --gpu-architecture=compute_35 --gpu-code=sm_35
$ nvcc code.cu -arch=sm_35
```

Stanford University

Profiling

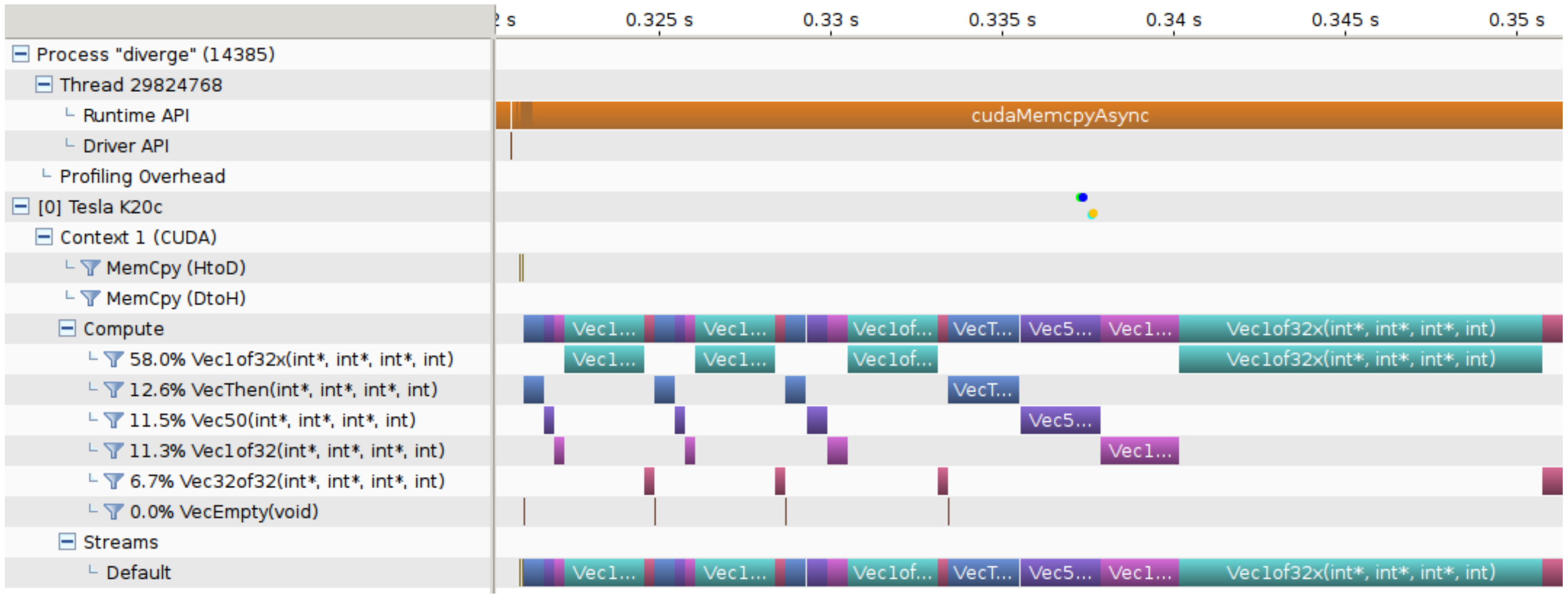
- The Visual Profiler (nvvp) is the simplest way to profile your code. You get a graphical interface that helps navigate the information.
- You can also use a command line tool (good for clusters): nvprof.
- This can also be used to collect information on a cluster to be visualized using nvvp on your local computer.

https://docs.nvidia.com/cuda/profiler-users-guide/index.html

Run

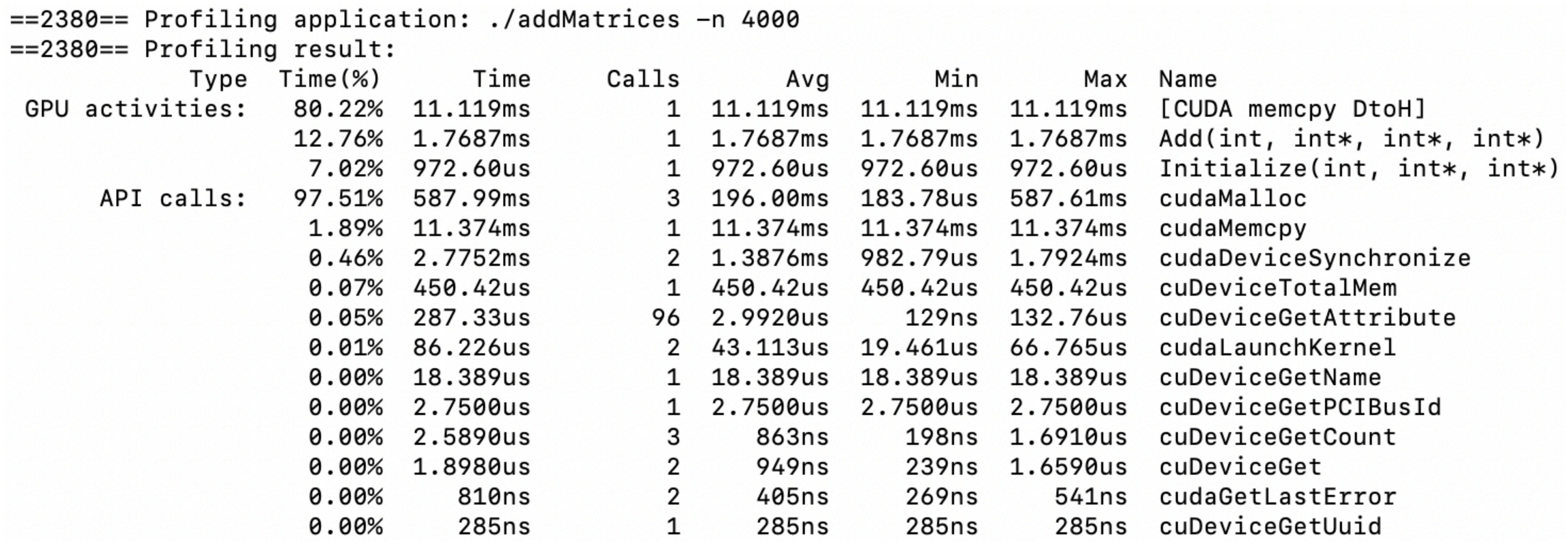
```
$ nvprof ./addMatrices -n 4000
```

Visual Profiler Timeline



Process "diverge" (14385)
Thread 29824768
Runtime API
Driver API
Profiling Overhead
[0] Tesla K20c
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
58.0% Vec1of32x(int*, int*, int*, int)
12.6% VecThen(int*, int*, int*, int)
11.5% Vec50(int*, int*, int*, int)
11.3% Vec1of32(int*, int*, int*, int)
6.7% Vec32of32(int*, int*, int*, int)
0.0% VecEmpty(void)
Streams
Default

Nvprof output



```
==2380== Profiling application: ./addMatrices -n 4000
==2380== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   80.22%  11.119ms         1  11.119ms  11.119ms  11.119ms  [CUDA memcpy DtoH]
                   12.76%  1.7687ms         1  1.7687ms  1.7687ms  1.7687ms  Add(int, int*, int*, int*)
                    7.02%  972.60us         1  972.60us  972.60us  972.60us  Initialize(int, int*, int*)
      API calls:   97.51%  587.99ms         3  196.00ms  183.78us  587.61ms  cudaMalloc
                    1.89%  11.374ms         1  11.374ms  11.374ms  11.374ms  cudaMemcpy
                    0.46%  2.7752ms         2  1.3876ms  982.79us  1.7924ms  cudaDeviceSynchronize
                    0.07%  450.42us         1  450.42us  450.42us  450.42us  cuDeviceTotalMem
                    0.05%  287.33us        96  2.9920us     129ns  132.76us  cuDeviceGetAttribute
                    0.01%  86.226us         2  43.113us  19.461us  66.765us  cudaLaunchKernel
                    0.00%  18.389us         1  18.389us  18.389us  18.389us  cuDeviceGetName
                    0.00%  2.7500us         1  2.7500us  2.7500us  2.7500us  cuDeviceGetPCIBusId
                    0.00%  2.5890us         3     863ns     198ns  1.6910us  cuDeviceGetCount
                    0.00%  1.8980us         2     949ns     239ns  1.6590us  cuDeviceGet
                    0.00%     810ns         2     405ns     269ns     541ns  cudaGetLastError
                    0.00%     285ns         1     285ns     285ns     285ns  cuDeviceGetUuid
```

cuda-memcheck

Several tools available to verify your code.

https://docs.nvidia.com/cuda/cuda-memcheck/index.html

4 tools accessible with **`cuda-memcheck`** command:

Memcheck	Memory access checking
Initcheck	Global memory initialization checking
Racecheck	Shared memory hazard checking
Synccheck	Synchronization checking

Usage examples:

```
$ cuda-memcheck ./memcheck_demo
$ cuda-memcheck --leak-check full ./memcheck_demo
```

GPU Optimization!

Top factors to consider

- Memory: data movement!
- Occupancy and concurrency:
  “hide latency through concurrency”
- Control flow: branching

Memory

- The number one bottleneck in scientific applications are not flops!
- Flops are plentiful and abundant.
- The problem is data starvation.
- Computing units are idle because they are waiting on data coming from / going to memory.

Memory access

- Similar to a regular processor but with some important differences.
- Caches are used to optimize memory accesses: L1 and L2 caches.
- Cache behavior is complicated and depends on the compute capability of the card (that is the type of GPU you are using).
- Generations (compute capability): 2.x, 3.x, 5.x, and 6.x (no 4.x).
- For this class, we have access to **3.7** hardware.
- We will focus on this for our discussion.

Cache

- There are two main levels of cache: L1 and L2.
- L2 is shared across SMs.
- Each L1 is assigned to only one SM.
- When reading data from global memory, the device can only read entire cache lines.
- There is no such thing as reading a single float from memory.
- To read a single float, you read the entire cache line and “throw out” the data you don’t need.

Cache lines

- L1: 128-byte lines = 32 floats
- L2: 32-byte segments = 8 floats

How is data read from memory then?

- Remember how the hardware works.
- All threads in a warp execute the same instruction at the same time.
- Take 32 threads in a warp.
- Each thread is going to request a memory location.
- Hardware groups these requests into a number of cache line requests.
- Then data is read from memory.
- At the end of the day, what matters is not how much data threads are requesting from memory, it’s how many memory transactions (cache lines) are performed!

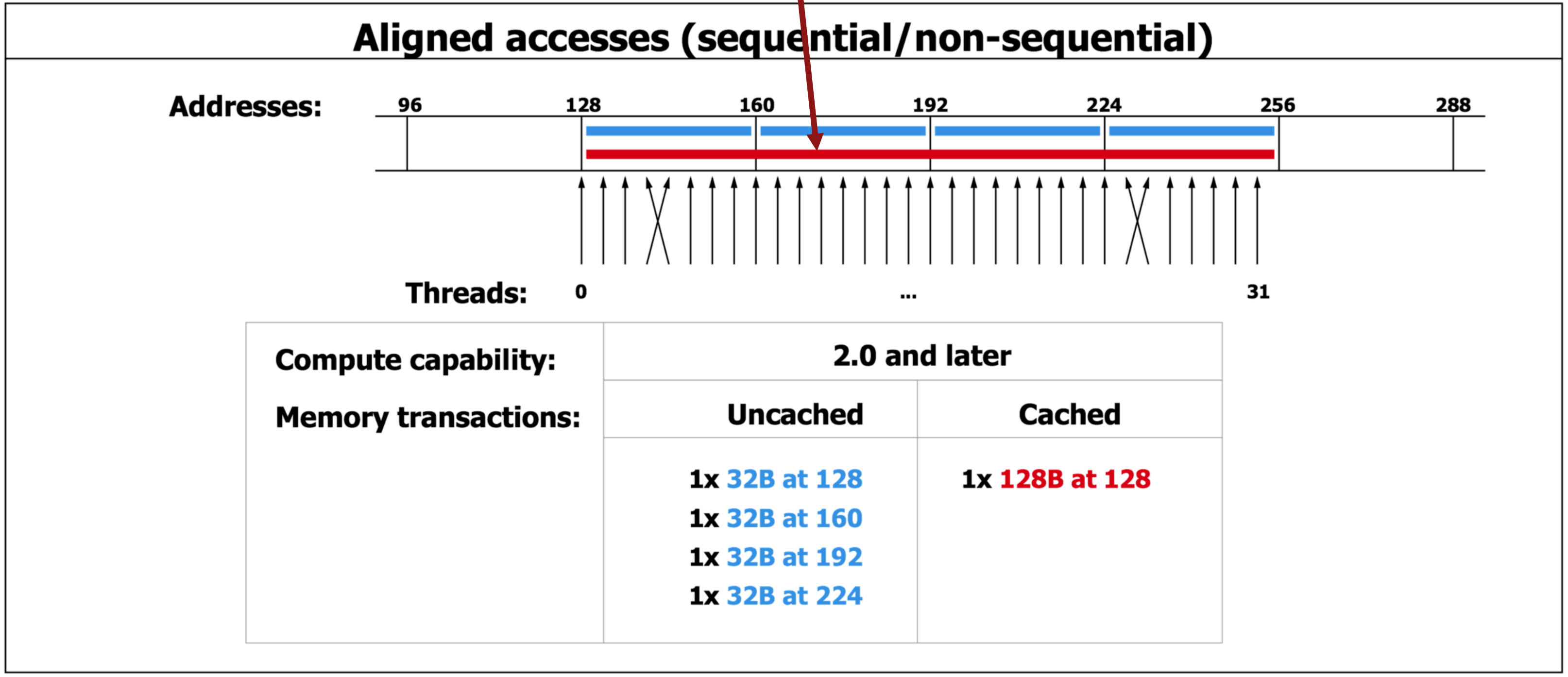
Example

```
__global__ void offsetCopy(float* odata, float* idata, int offset) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```

`offset = 0`

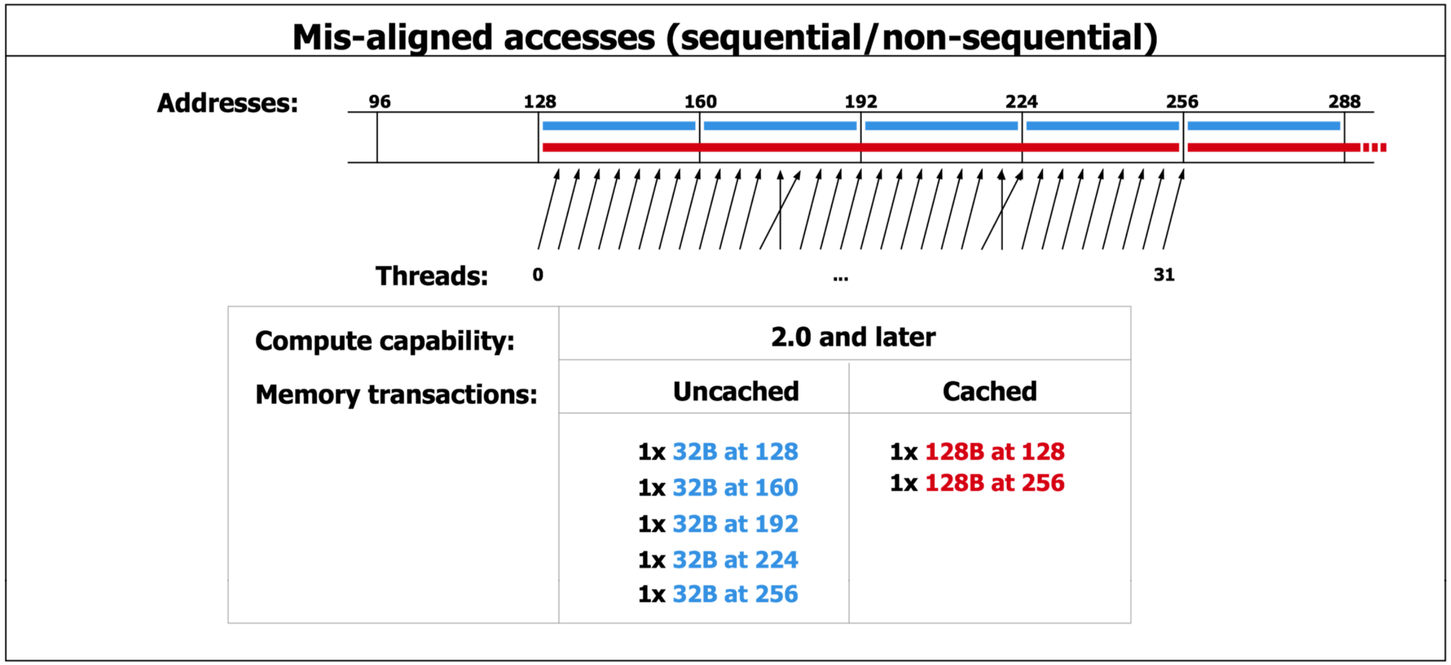
This is the best case.
All memory accesses are coalesced into a single memory transaction.

One memory transaction



Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Misaligned by 1

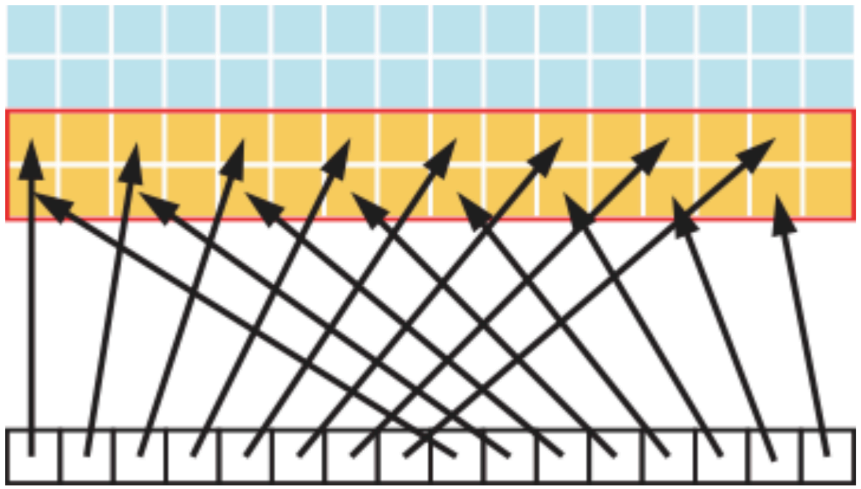


Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

```
offset = 1
```

Strided access

```
__global__ void stridedCopy(float* odata, float* idata, int stride) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[stride*xid];
}
```



Stride is 2.
Effective bandwidth of application reduced by 2.

Efficiency of reads

Perfectly coalesced access: one memory transaction

  Bandwidth = Peak Bandwidth

Misaligned: two memory transactions

  Bandwidth = Peak Bandwidth / 2

Strided access

  Bandwidth = Peak Bandwidth / s
  for a stride of s

Random access

  Bandwidth = Peak Bandwidth / 32

```
__global__ void randomCopy(float* odata, float* idata, int* addr) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[addr[xid]];
}
```

Summary of main features of device memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached in L1 and L2 by default on devices of compute capability 2.x; cached only in L2 by default on devices of higher compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

†† Cached in L1 and L2 by default on devices of compute capability 2.x and 3.x; devices of compute capability 5.x cache locals only in L2.

Data type size

- The previous discussion covered the most common case where threads read a 4-byte float or int.
- char: 1 byte. float: 4 bytes. double: 8 bytes.
- CUDA has many built-in data types that are shorter/longer:

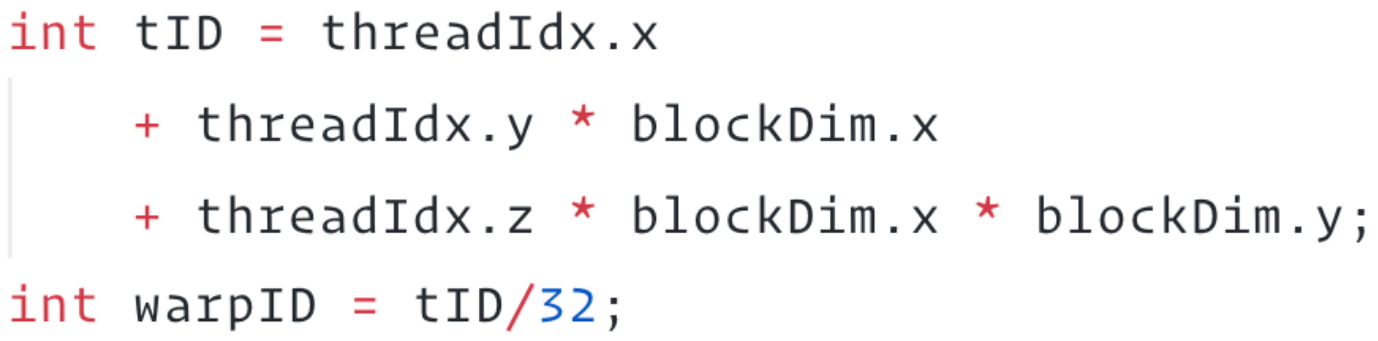
```
    char1-4, uchar1-4
    short1-4, ushort1-4
    int1-4, uint1-4
    long1-4, ulong1-4
    longlong1-2, ulonglong1-2
    float1-4
    double1-2
```

- Example: `char1, int2, long4, double2`

How to calculate the warp ID

- How do you know if 2 threads are in the same warp?
- Answer: look at the thread ID. Divide by 32: you get the warp ID.
- Thread ID is computed from the 1D, 2d or 3D thread index using:

$$x + yD_x + zD_xD_y$$



```
int tID = threadIdx.x
        + threadIdx.y * blockDim.x
        + threadIdx.z * blockDim.x * blockDim.y;
int warpID = tID/32;
```