

CME 213, ME 339—Spring 2021

Eric Darve, ICME, Stanford



“There are two ways to write error-free programs; only the third one works.” (Alan J. Perlis)

Let's get started!

How to transfer the files

- scp
- sshfs; [Sherlock instructions](#)

Demo

- `ssh darve@icme-gpu.stanford.edu`
- You have to use VPN if you are off-campus.

Make sure the CUDA library and compiler are loaded.

- `module avail`
- `module list`
- `module load cuda`
- `module show cuda`
- `module unload cuda`
- `module purge`

Demo

Recommended: add

```
# loading CUDA modules  
module load cuda
```

at the end of `.bashrc` in your HOME directory. This will load the module you need when you log in.

You will find that it is annoying to repeatedly log in on the cluster and provide your password and complete the dual authentication process.

A useful command is `screen`.

It allows running multiple shells.

To start screen: `$ screen`. Then use the following shortcuts:

Shortcut	Command
<code>Ctrl+a c</code>	Create a new window (with shell)
<code>Ctrl+a "</code>	List all window
<code>Ctrl+a 0</code>	Switch to window 0 (by number)
<code>Ctrl+a </code>	Split current region vertically into two regions
<code>Ctrl+a S</code>	Split current region horizontally into two regions
<code>Ctrl+a tab</code>	Switch the input focus to the next region
<code>Ctrl+a X</code>	Delete window but keep shell
<code>Ctrl+a Ctrl+a</code>	Toggle between the current and previous region
<code>Ctrl+a k</code>	Close the current shell
<code>Ctrl+a \</code>	Close all shells
<code>Ctrl+a ?</code>	Help

More advanced commands

Basic SLURM commands

sinfo

Demo

```
sbatch script.sh
```

```
#!/bin/bash  
#SBATCH -o job_%j.out  
#SBATCH -p CME  
#SBATCH --gres=gpu:1
```

Batch submission:

```
sbatch script.sh; queue
```

Demo

Blocking command:

```
srun -p CME --gres=gpu:1 ./deviceQuery
```

or

```
srun -o slurm.sh.out -p CME --gres=gpu:1 ./deviceQuery
```

or

```
srun -p CME --gres=gpu:1 ./script.sh
```

What not to do!

```
srun -p CME --gres=gpu:1 --pty /bin/bash
```

This reserves a node for you "indefinitely."

Demo

Controlling the number of GPUs you have access to:

```
srun -p CME --gres=gpu:2 ./deviceQuery
```

```
srun -p CME --gres=gpu:3 ./deviceQuery
```

Device 0: "Quadro RTX 6000"

CUDA Driver Version / Runtime Version	11.0 / 11.0
CUDA Capability Major/Minor version number:	7.5
Total amount of global memory:	24220 MBytes (25396838400 bytes)
(72) Multiprocessors, (64) CUDA Cores/MP:	4608 CUDA Cores
L2 Cache Size:	6291456 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	65536
Warp size:	32
Maximum number of threads per multiprocessor:	1024
Maximum number of threads per block:	1024
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(2147483647, 65535, 65535)

Other SLURM commands

- `squeue`
- `scancel`

Let's run

```
./bandwidthTest
```

[CUDA Bandwidth Test] - Starting...

Running on...

Device 0: Quadro RTX 6000

Quick Mode

Host to Device Bandwidth, 1 Device(s)

PINNED Memory Transfers

Transfer Size (Bytes)	Bandwidth(GB/s)
32000000	12.6

Device to Host Bandwidth, 1 Device(s)

PINNED Memory Transfers

Transfer Size (Bytes)	Bandwidth(GB/s)
32000000	13.2

Device to Device Bandwidth, 1 Device(s)

PINNED Memory Transfers

Transfer Size (Bytes)	Bandwidth(GB/s)
32000000	541.0

Result = PASS

firstProgram.cu

```
checkCudaErrors(...)
```

CUDA functions often fail silently

Use this to check for errors before continuing

```
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);  
  
cudaFree(d_output);
```



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

N : number of threads to launch for function `kernel`

Threads are numbered 0 to $N - 1$.

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

global / host / device

???

`__global__` kernel will be

- Executed on the device
- Callable from the host

`__host__` kernel will be

- Executed on the host
- Callable from the host

`__device__` kernel will be

- Executed on the device
- Callable from the device only

Get information about the current thread

Use the built-in variable `threadIdx`

We will learn more about this later

Run

```
darve@icme-gpu:~/Lecture_08$ srun -p CME --gres=gpu:1 ./firstProgram
Entry          0, written by thread    0
Entry          9, written by thread    3
...
Entry        961, written by thread   31
```



```
darve@icme-gpu:~/Lecture_08$ srun -p CME --gres=gpu:1 ./firstProgram -N=1024
Using 1024 threads = 32 warps
Entry      0, written by thread      0
Entry    10404, written by thread   102
Entry    41616, written by thread   204
...
Entry    842724, written by thread   918
Entry  1040400, written by thread  1020
Entry  1046529, written by thread  1023
```

```
darve@icme-gpu:~/Lecture_08$ srun -p CME --gres=gpu:1 ./firstProgram -N=1025
CUDA error at firstProgram.cu:48 code=9
      (cudaErrorInvalidConfiguration) "cudaGetLastError()"
Using 1025 threads = 33 warps
srun: error: icmet01: task 0: Exited with exit code 1
```

!!!

Let's consult the Quadro RTX 6000 data sheet

[Quadro RTX 6000](#)

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

N cannot be greater than 1,024.

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

What we need is

```
kernel<<<num_blocks, block_size>>>(d_output);
```

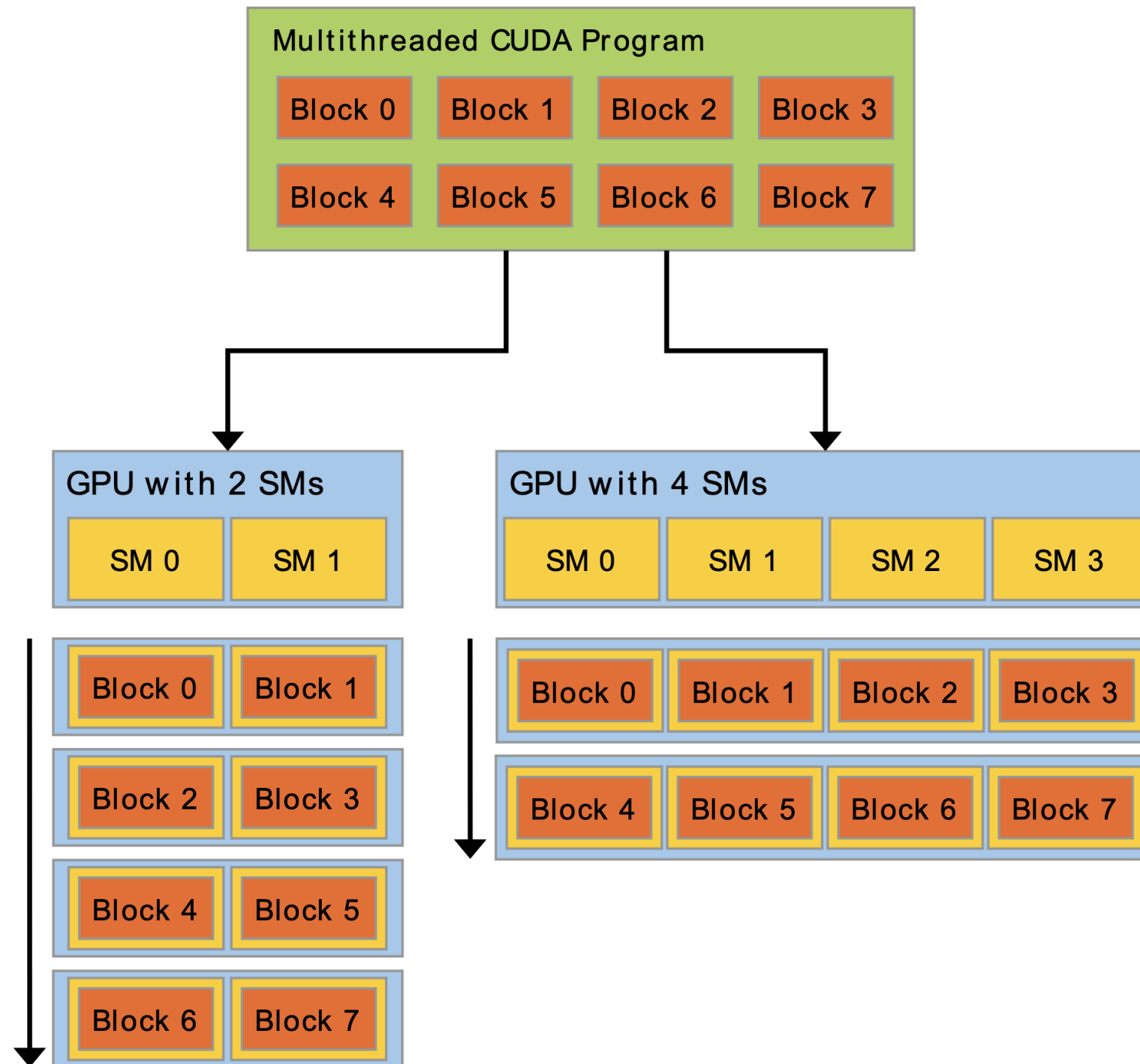
```
kernel<<<num_blocks, block_size>>>(d_output);
```

`block_size` can be at most 1,024

Use more blocks!

Calculation should be organized into:

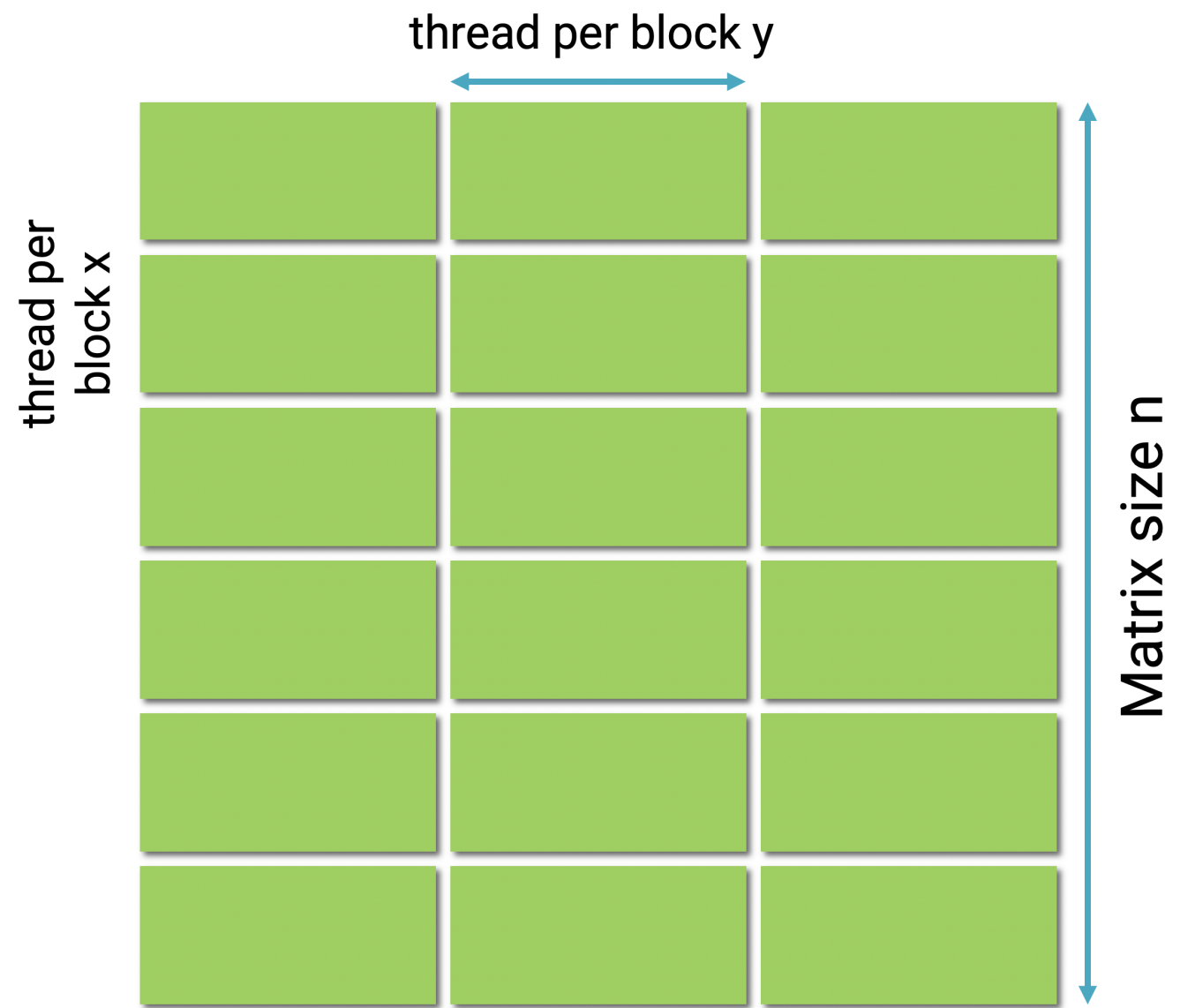
- blocks that fit on each SM (limited number of threads)
- several blocks forming a grid (so that an "unlimited" number of threads can be used)



Defining dimensions

```
dim3 block_size(Nx);  
dim3 num_blocks(Mx);  
  
dim3 block_size(Nx, Ny);  
dim3 num_blocks(Mx, My);  
  
dim3 block_size(Nx, Ny, Nz);  
dim3 num_blocks(Mx, My, Mz);  
  
kernel<<<num_blocks, block_size>>>(d_output);
```

Let's use this to write a program to add two matrices.



```
dim3 th_block(32,n_thread/32);

int blocks_per_grid_x = (n + th_block.x - 1) / th_block.x;
int blocks_per_grid_y = (n + th_block.y - 1) / th_block.y;

dim3 num_blocks(blocks_per_grid_x, blocks_per_grid_y);

Add<<<num_blocks, th_block>>>(n, d_a, d_b, d_c);
```

Math formula for number of blocks

$$\begin{aligned} \text{num_blocks} = & \\ (\text{num_threads_total} + \text{num_thread_per_block} - 1) & \\ / \text{num_thread_per_block} & \end{aligned}$$

Try out with

$$\begin{aligned} \text{num_threads_total} &= 5 \\ \text{num_thread_per_block} &= 4 \end{aligned}$$

```
__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];  
    }  
}
```

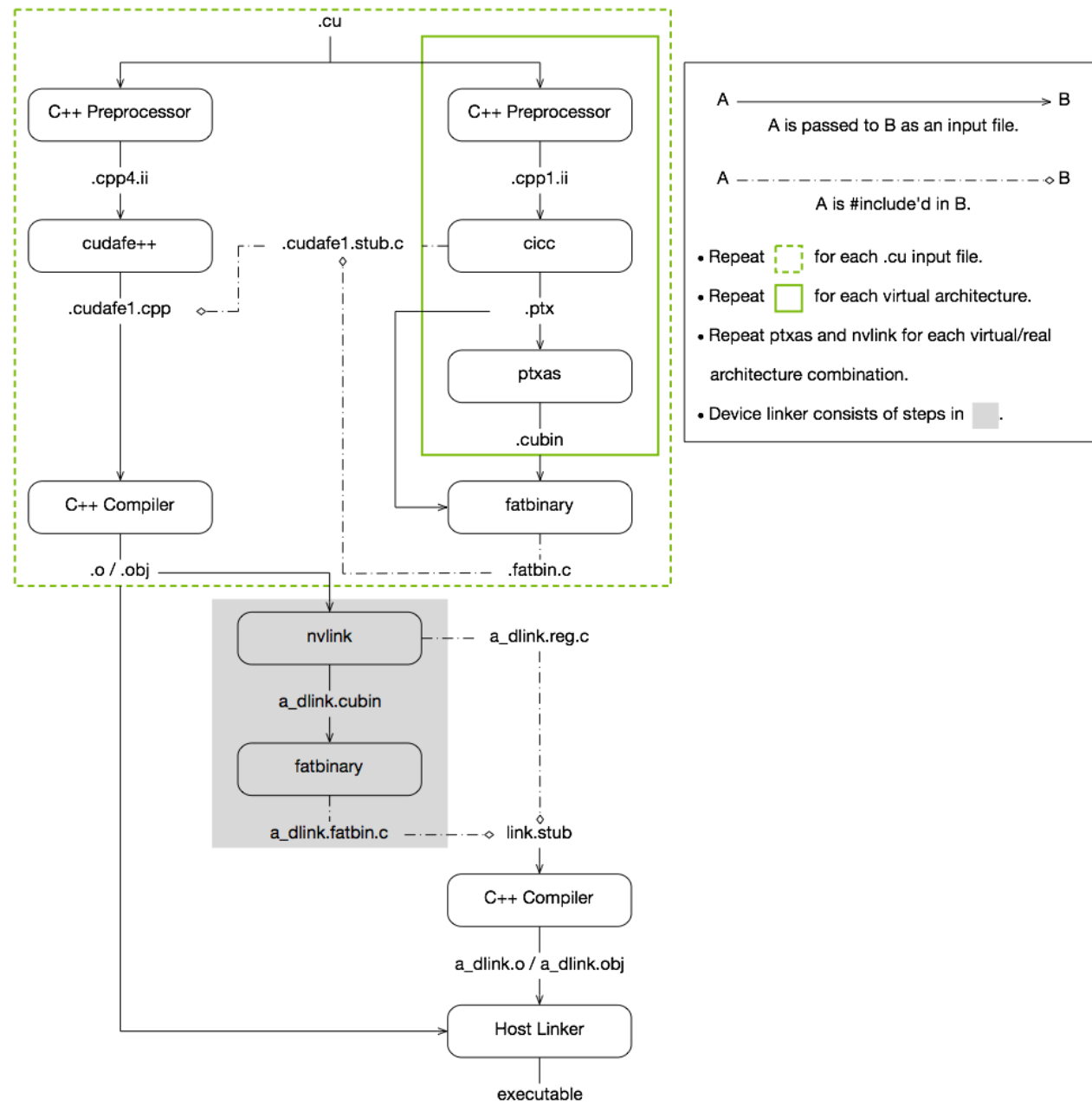
Built-in variable	Description
threadIdx	thread index in block
blockDim	number of threads in a block
blockIdx	block index in grid
gridDim	number of blocks in grid
warpSize	number of threads in a warp

STL vector cannot be used with CUDA.

CUDA has its own mechanism to allocate and manage memory.

See [Thrust](#) for an STL like vector implementation in CUDA.

Compiling CUDA code



Most CPUs offer binary code compatibility and rely on a published instruction set architecture.

A given compiled code can run on many different processors.

The situation is different with GPUs.

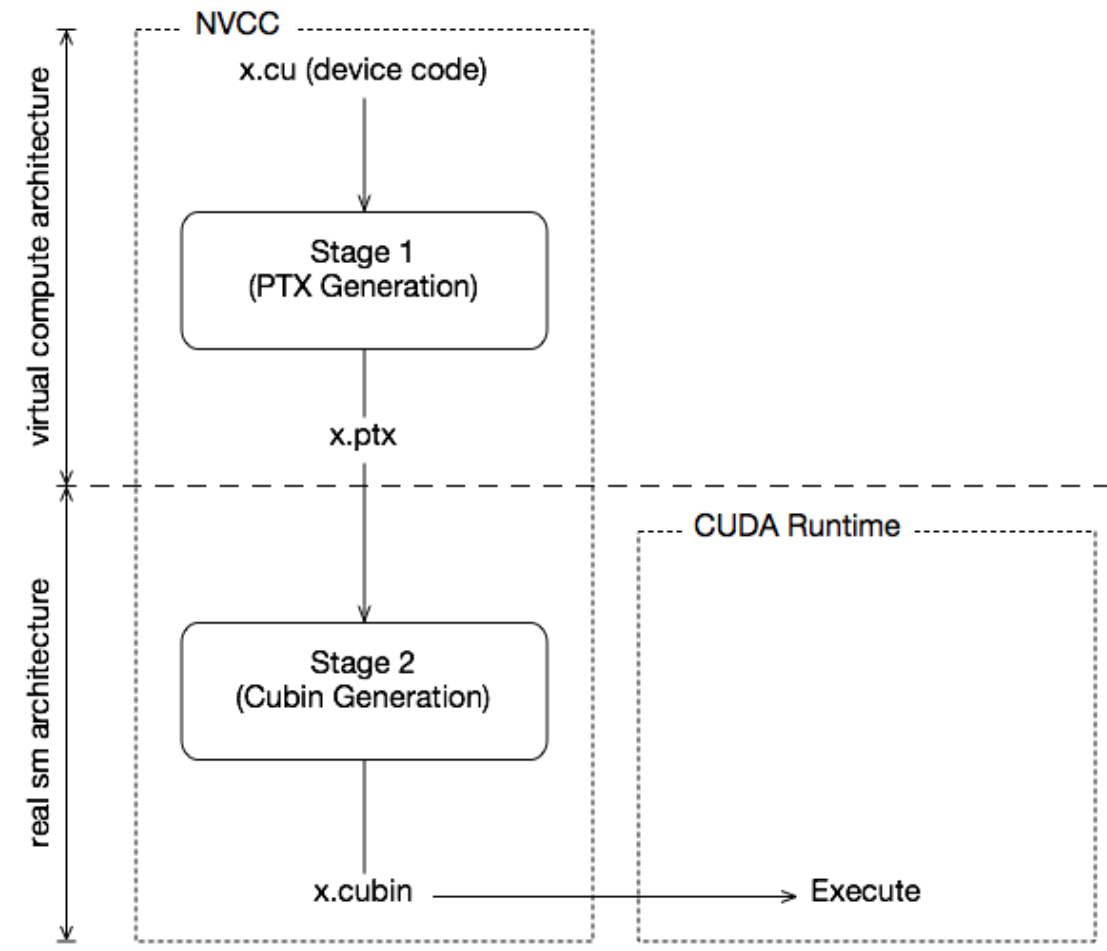
GPU improvements mean that binaries for different processors are incompatible.

Compilation happens in two stages:

1. Code for virtual architecture is generated; PTX
2. Code for real architecture is generated

PTX assembly code relies on a specific set of features or GPU capabilities

Real architecture: binary code that can be executed on a given GPU



File extension	Description
.cu	CUDA source file
.ptx	PTX intermediate assembly file
.cubin	CUDA device code binary file
.fatbin	CUDA fat binary file that may contain multiple PTX and CUBIN files

When compiling:

- one virtual architecture is chosen
- some (or none) real architectures are specified

If a real architecture is compiled and matches the GPU, the binary is loaded and runs!

If a real architecture for the GPU is missing, a matching GPU binary code is generated when the application is launched using the PTX code.

This is called `just-in-time` compilation.

Virtual architecture names start with compute_

Real architecture names start with sm_

Example

```
nvcc a.cu --gpu-architecture=compute_50 --gpu-code=sm_50,sm_75
```

Use virtual architecture compute_50

Generate code for two GPUs: sm_50, sm_75

On icme-gpu, try

```
--gpu-architecture=compute_50 --gpu-code=sm_50
```

```
--gpu-architecture=compute_50 --gpu-code=sm_50
```

Fails because our GPU is sm_75

Try

```
--gpu-architecture=compute_50 --gpu-code=sm_75
```

Success

```
--gpu-architecture=compute_75 --gpu-code=sm_50
```

Fails because sm_50 does not support compute_75 features

Note on `--gpu-architecture`

`--gpu-architecture` alone does not trigger assembly of the corresponding PTX.

That is the role of `--gpu-code`.

Try

```
--gpu-architecture=compute_50 --gpu-code=compute_50,sm_50
```

Succeeds

Why?

Wrong sm_50; but PTX for compute_50 is loaded

Can be JIT compiled for sm_75

Win!

```
--gpu-architecture=compute_75 --gpu-code=sm_75
```

Compile just for our GPU

shorthands

`--gpu-architecture=compute_75`

is equivalent to

`--gpu-architecture=compute_75 --gpu-code=compute_75`

Only generate and embed PTX

JIT required for all GPUs

`--gpu-architecture=sm_75` (a real architecture option)

is equivalent to

`--gpu-architecture=compute_75 --gpu-code=compute_75,sm_75`

Generate binary for `sm_75` + PTX for JITs on GPUs that support `compute_75`

Recommended

```
--gpu-architecture=compute_75 --gpu-code=sm_75
```

Shorter option (which embeds the PTX with the binary)

```
--gpu-architecture=sm_75 or -arch=sm_75
```

[List of virtual architectures](#)

[List of real architectures](#)

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

`nvcc --help`

The old way:

[Visual Profiler](#)

The new way:

[NVIDIA Nsight Systems](#) for GPU and CPU sampling and tracing

[NVIDIA Nsight Compute](#) for GPU kernel profiling



CUDA-MEMCHECK

Tool	Description
memcheck	Memory access error and leak detection
racecheck	Shared memory data access hazard detection
initcheck	Unitialized device global memory access detection
synccheck	Thread synchronization hazard detection