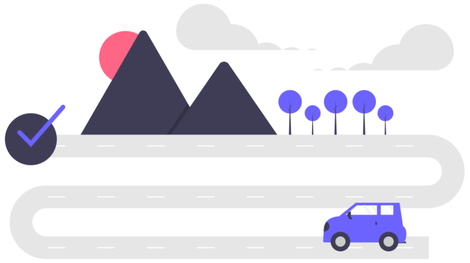
CME 213, ME 339 Spring 2023
Introduction to parallel computing using MPI, openMP, and CUDA

Eric Darve, ICME, Stanford

"You can either have software quality or you can have pointer arithmetic, but you cannot have both at the same time." (Bertrand Meyer)

Stanford University

Recap



- General purpose GPU computing
- GPU architecture vs multicore-processor
- Streaming multiprocessors, thread blocks, thread warps
- SIMT vs SIMD processor architectures
- Basic organization of a GPU code

Compute capability

Compute capability of GPU

- The compute capability of a device is represented by a version number, also sometimes called its “SM version.”

- This version number identifies the **features** supported by the GPU hardware and is used by applications at runtime to determine which **hardware features and/or instructions** are available on the present GPU.

- This is important to know when programming and to understand the CUDA documentation.

GPU generations

Micro-architecture	Release	Compute Capability	GPU code name	Consumer/Pro
G70	2005			
Tesla	2006	1.0-1.3	GXX, GT2XX	
Fermi	2010	2.0-2.1	GFXXX	
Kepler	2012	3.0-3.7	GKXXX	
Maxwell	2014	5.0–5.3	GMXXX	
Pascal	2016	6.0-6.2	GPXXX	Consumer
Volta	2017	7.0-7.2	GVXXX	Professional
Turing	**2018**	**7.5**	**TUXXX**	**Consumer**
Ampere	2020	8.0-8.6	GAXXX	
Ada Lovelace	2022	8.9	ADXXX	Consumer
Hopper	2022	9.0	GHXXX	Professional
Blackwell	2024	?	?	?

Achieving performance on a GPU

Roadmap for performance on a GPU

This is a high level view. We will discuss this in more details in future lectures.

Performance requires mapping the algorithm to the architecture of the processor.

1. Amount of thread level **parallelism is massive;** high amount of parallel work is required

2. All threads should execute the same instruction; avoid **branching**

3. **Memory access** should follow a specific pattern, which we will cover

4. Although cache is small, it is key to **reducing memory traffic.** Three levels of cache: L2, L1, shared memory.

Let's get started with GPU programming on icme-gpu!

Steps

- Log on computer and transfer files.
- You will need to load the appropriate libraries.

modules

```
darve@icme-gpu1:~$ module avail

------------------------------------------ /apps/software/modules/Core -------------------------------------------
   Gurobi/9.0.3        cuda/11.7   (g)        cudnn/7.6.2   (g,D)    openmpi/4.1.1 (D)
   clang/10.0          cuda/11.8   (g,L,D)    julia/1.1.1            rclone/1.55.0
   cuda/11.4    (g)    cudnn/7.4.1 (g)        openmpi/4.1.0

  Where:
   g:  built for GPU
   L:  Module is loaded
   D:  Default Module

Use "module spider" to find all possible modules.
Use "module keyword key1 key2 ..." to search for all possible modules matching any of the "keys".
```

You need to load the appropriate modules to access the resources you need.

.bashrc

Add the following to `~/.bashrc`

```
module load cuda
module load openmpi
```

VSCode remote shell bug

- When using a remote shell with VSCode, the file `/etc/profile` is not run.
- So you need to modify your `~/.bashrc` to make sure `/etc/profile` is executed. This ensures that the `module` command works correctly.
- Add at the top of `~/.bashrc:`

```
module &>/dev/null

if [ "$?" == "127" ]; then

  . /etc/profile

fi
```

screen

- A useful command to avoid having to open multiple shell windows is `screen`.
- It allows running multiple shells after logging only once.

How to use screen

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

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

https://linuxize.com/post/how-to-use-linux-screen/

Running code on a GPU

- You can use a batch script with SLURM.
- Example: `script.sh`

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

./gpu_code
```

```
$ sbatch script.sh; squeue
```

Blocking command

- The previous command runs the script as a batch.
- This means that the command returns immediately and the output is saved in a file.
- Instead you can run a blocking command:

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

- This is the same thing as a batch script, but instead the command blocks and the output shows up on the screen.

Demo

```
$ make clean && make

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

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

What not to do…

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

This reserves a node for you “indefinitely.”

```
darve@icmet01:~/2023/CUDA$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
              8418       CME     bash    darve  R       0:08      1 icmet01
darve@icmet01:~/2023/CUDA$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
k80          up    6:00:00      5   idle icme[01-05]
CME          up    2:00:00      1    mix icmet01
CME          up    2:00:00      5   idle icmet[02-06]
V100         up    8:00:00      1   idle icme06
darve@icmet01:~/2023/CUDA$
```

Running on multiple GPUs

Requesting 4 GPUs with SLURM:

```
$ srun -p CME --gres=gpu:4 ./deviceQuery
```

Output of `deviceQuery`

```
darve@icme-gpu1:~/2023/CUDA$ srun --gres=gpu:1 -p CME ./deviceQuery
/home/darve/2023/CUDA/./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Quadro RTX 6000"
  CUDA Driver Version / Runtime Version          11.8 / 11.8
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 22697 MBytes (23799136256 bytes)
  (72) Multiprocessors, ( 64) CUDA Cores/MP:     4608 CUDA Cores
  GPU Max Clock rate:                            1770 MHz (1.77 GHz)
  Memory Clock rate:                             7001 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 6291456 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 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)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 193 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.8, CUDA Runtime Version = 11.8, NumDevs = 1
Result = PASS
darve@icme-gpu1:~/2023/CUDA$
```

Output of `bandwidthTest`

```
darve@icme-gpu1:~/2023/CUDA$ srun --gres=gpu:1 -p CME ./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                     465.1

Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
darve@icme-gpu1:~/2023/CUDA$
```

Our first program

See file `firstProgram.cu`

Memory allocation

```
cudaMalloc(&d_output, sizeof(int) * N)
```

Allocates memory on the GPU

`d_output` contains the address of a memory location on the GPU memory

Error checking

- Many functions on GPUs run **asynchronously.**
- This makes error-checking difficult.
- The only way to check for asynchronous errors just after some asynchronous function call is to synchronize after the call by calling **`cudaDeviceSynchronize()`**.
- Then, we can check the error code returned by `cudaDeviceSynchronize()`.
- The runtime maintains an error variable that is initialized to `cudaSuccess` and is overwritten by the error code every time an error occurs.
- **`cudaPeekAtLastError()`** returns this variable.
- **`cudaGetLastError()`** returns this variable and resets it to `cudaSuccess`.

Memory allocation error

```
checkCudaErrors(cudaMalloc(&d_output, sizeof(int) * N));
```

The function `checkCudaErrors` is defined in `utils.h` and checks if `cudaMalloc` returned an error.

Calling a kernel

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

- Calling a kernel function.
- Number of blocks = 1
- Number of threads in block = N

Asynchronous execution

- The kernel runs asynchronously.
- This means that the CPU function returns immediately, and the execution gets deferred.
- If the kernel fails, the CPU code keeps running. The failure is **silent.**
- This leads to strange bugs where the code seems to complete normally, but the result is incorrect.
- So you should always check that the kernel terminates with no error message.
- This requires 2 steps:

1. `cudaDeviceSynchronize();`
2. `checkCudaErrors(cudaGetLastError());`

Retrieving data on the GPU

```
checkCudaErrors(cudaMemcpy(  &h_output[0], d_output, sizeof(int) * N,
                             cudaMemcpyDeviceToHost));
```

Copy back data from GPU to CPU.

CUDA kernel

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

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

- This is the function being run on the GPU.
- Each thread runs the function.
- Each thread gets a unique `threadIdx.x`

global / host / device

`__global__`	`__host__`	`__device__`
Executed on the device	Executed on the host	Executed on the device
Callable from the host	Callable from the host	Callable from the device

```
darve@icme-gpu1:~/2023/CUDA$ srun -p CME --gres=gpu:1 ./firstProgram -N=1024
Using 1024 threads = 32 warps
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from CUDA
[ RUN      ] CUDA.assign
Entry          0, written by thread     0
Entry      10404, written by thread   102
Entry      41616, written by thread   204
Entry      93636, written by thread   306
Entry     166464, written by thread   408
Entry     260100, written by thread   510
Entry     374544, written by thread   612
Entry     509796, written by thread   714
Entry     665856, written by thread   816
Entry     842724, written by thread   918
Entry    1040400, written by thread  1020
Entry    1046529, written by thread  1023
[       OK ] CUDA.assign (141 ms)
[----------] 1 test from CUDA (141 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (141 ms total)
[  PASSED  ] 1 test.
darve@icme-gpu1:~/2023/CUDA$ srun -p CME --gres=gpu:1 ./firstProgram -N=1025
Using 1025 threads = 33 warps
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from CUDA
[ RUN      ] CUDA.assign
CUDA error at firstProgram.cu:45 code=9(cudaErrorInvalidConfiguration) "cudaGetLastError()"
srun: error: icmet01: task 0: Exited with exit code 1
darve@icme-gpu1:~/2023/CUDA$
```

Let's consult the book of wisdom…

… the Quadro RTX 6000 data sheet

```
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

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
  GPU Max Clock rate:                            1770 MHz (1.77 GHz)
  Memory Clock rate:                             7001 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 6291456 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 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)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 193 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.0, CUDA Runtime Version = 11.0, NumDevs = 1
Result = PASS
```

What we need…

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

What we need is

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

Grid and thread 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

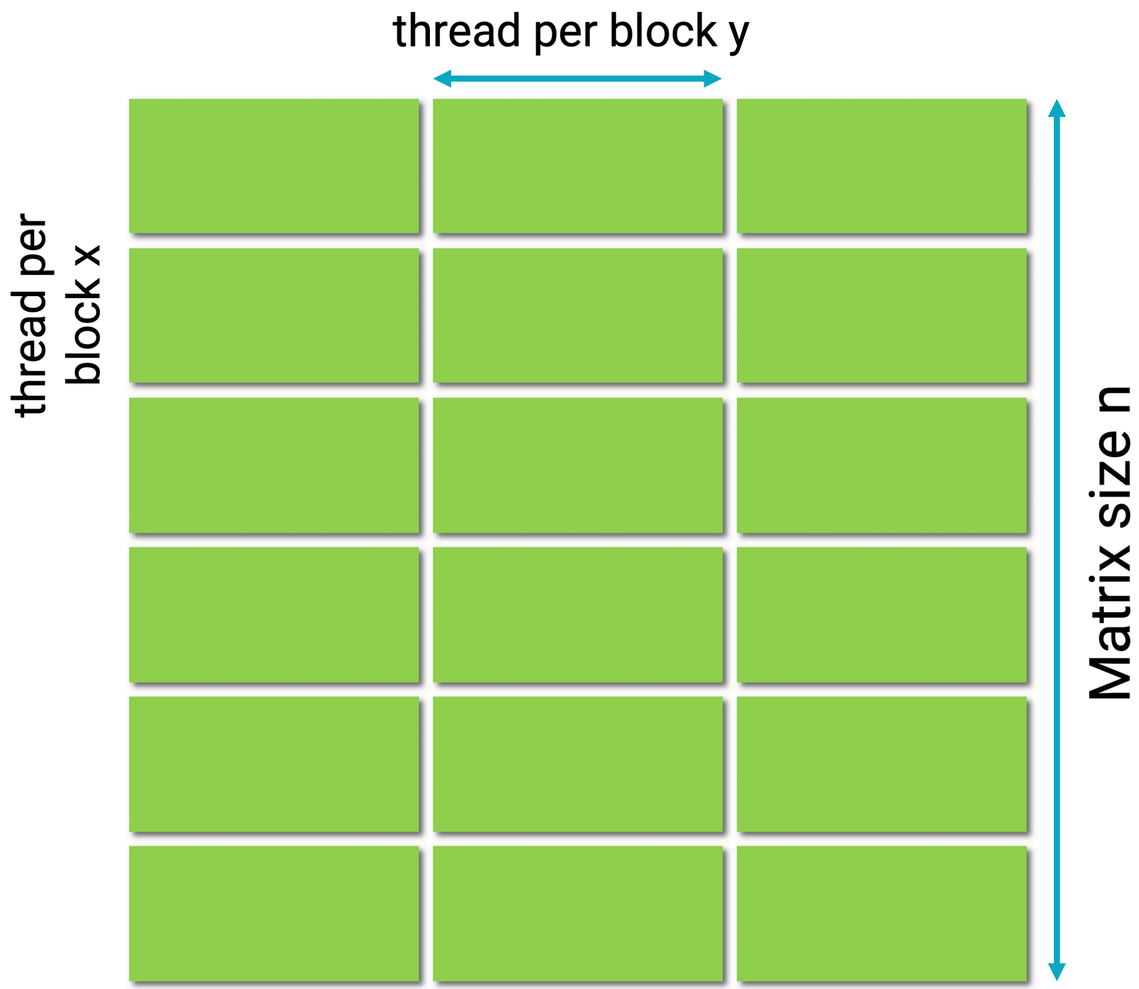
Dimensions

They are defined using `dim3`. Examples:

```
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);
```

Adding two matrices



We use a 2D indexing

`(n + th_block.x - 1) / th_block.x`

Formula is needed to make sure we have enough blocks and threads when `th_block.x` does not divide `n`.

```
int* d_a, *d_b, *d_c;

/* Allocate memory */
checkCudaErrors(cudaMalloc(&d_a, sizeof(int) * n*n));
checkCudaErrors(cudaMalloc(&d_b, sizeof(int) * n*n));
checkCudaErrors(cudaMalloc(&d_c, sizeof(int) * n*n));

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;
/* This formula is needed to make sure we process all entries in matrix */
dim3 num_blocks(blocks_per_grid_x, blocks_per_grid_y);

Initialize<<<num_blocks, th_block>>>(n, d_a, d_b);
cudaDeviceSynchronize();
checkCudaErrors(cudaGetLastError());

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

Kernel

- Row index: `i`
- Column index: `j`
- Block ID: `blockIdx`
- Block dimension: `blockDim`
- Thread ID in block: `threadIdx`
- Each thread runs `Add` once.
- **`if` required to avoid out of bound memory access.**

```
__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

C++ Standard Template Library

- C++ Standard Library is available in CUDA in different forms.
- libcu++ CUDA C++ standard library, documentation; provides a heterogeneous implementation of the C++ Standard Library that can be used in and between CPU and GPU code. Library is incomplete.
- Thrust CUDA C++'s high-productivity general-purpose library and parallel algorithms implementation.
- CUB CUDA C++'s high-performance collective algorithm toolkit