High Energy Physics

Traineeships in Advanced Computing for High Energy Physics (TAC-HEP)

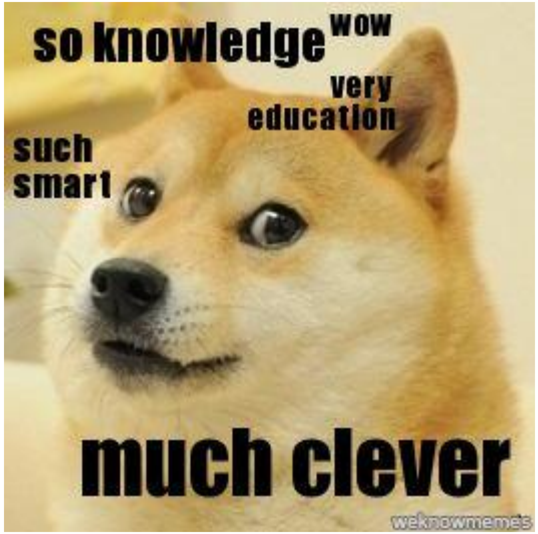
GPU programming module

Week 3 : Introduction to CUDA

Lecture 6 - September 26th 2024

What we learnt in the previous lecture

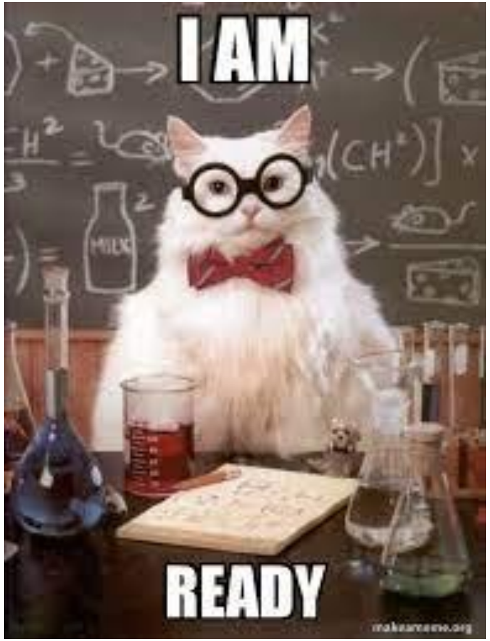
- Learnt about the Nvidia GPU architecture and explored the GPU characteristics
- Learnt about threads / blocks / grid
- Discussed about the CUDA core syntax
- Wrote our first “Hello world” CUDA kernel



Today

Today we will learn about :

- Basic memory management
- More on synchronization
- Error handling



Memory management

The CUDA programming model

In the previous lecture we learnt about the three main steps of a CUDA program :

What about these steps ?

- Copy the input data from CPU or host memory to the device memory
- Execute the CUDA program
- Copy the results from device memory to host memory

We were able to run our first "Hello World" CUDA program

1. Copy data from host to device

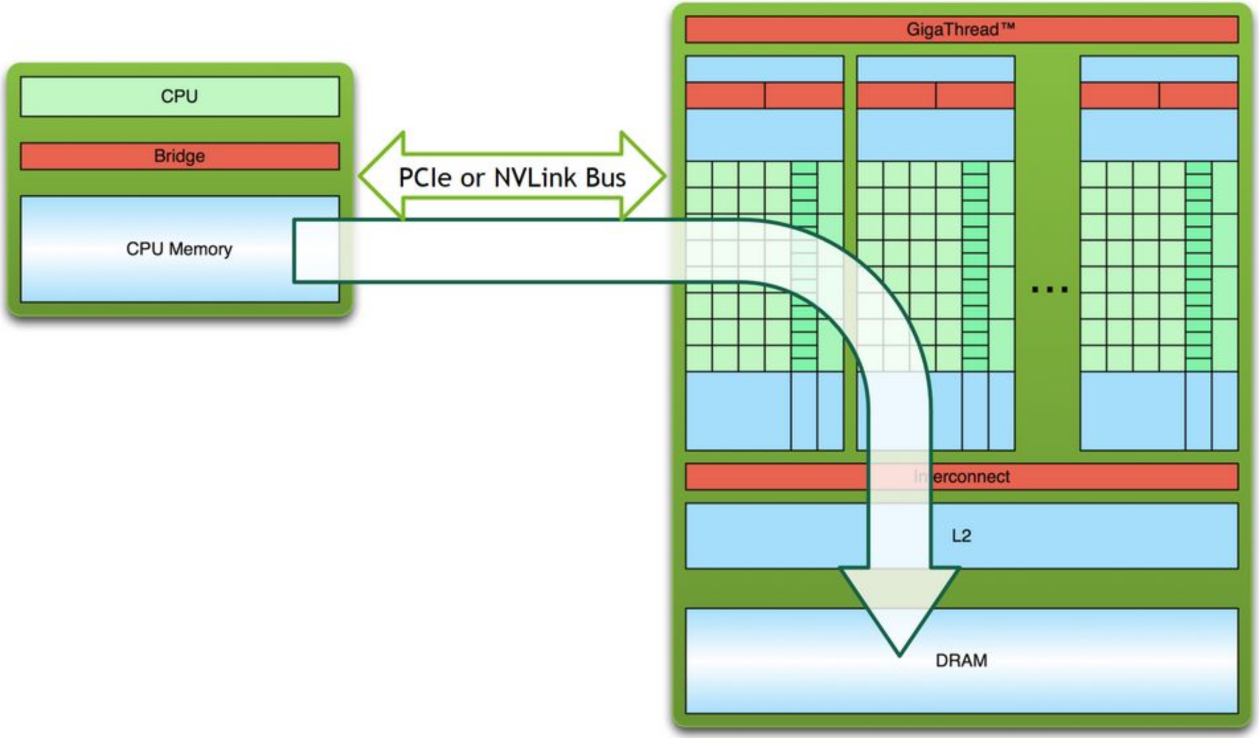


Image source [1]

2. Execute the CUDA program

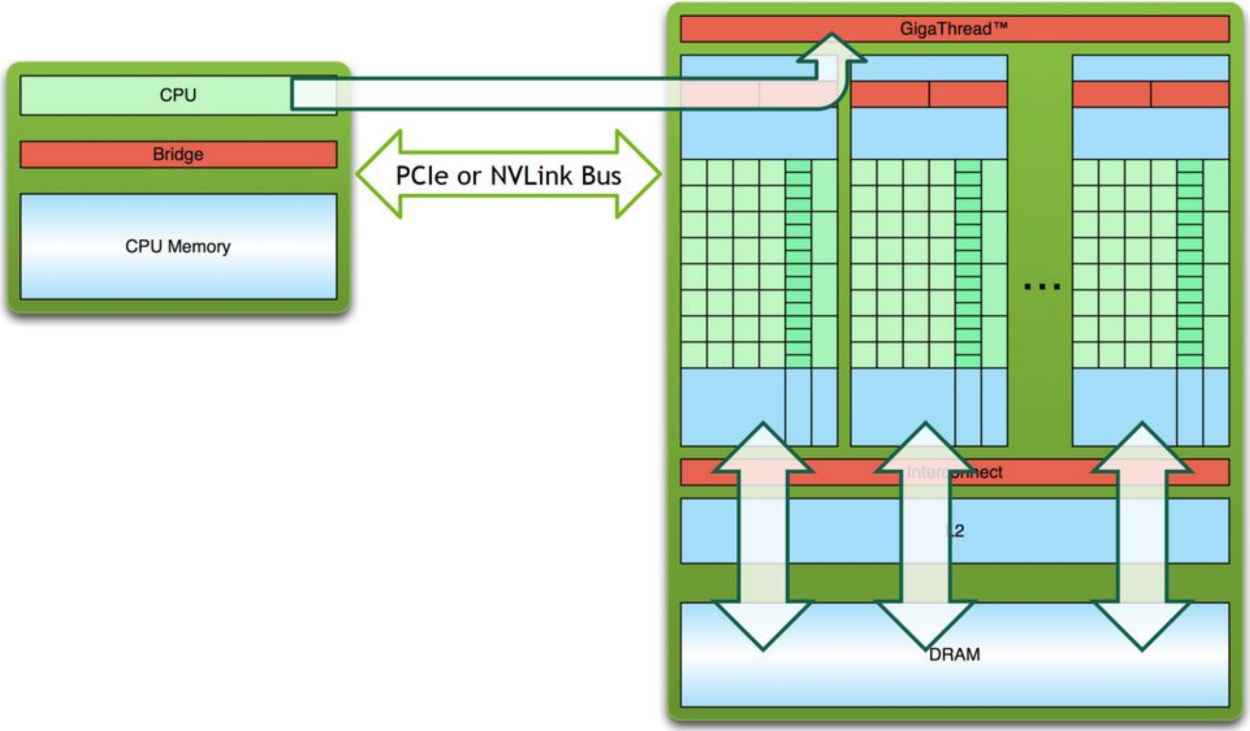


Image source [1]

3. Copy data from device back to host

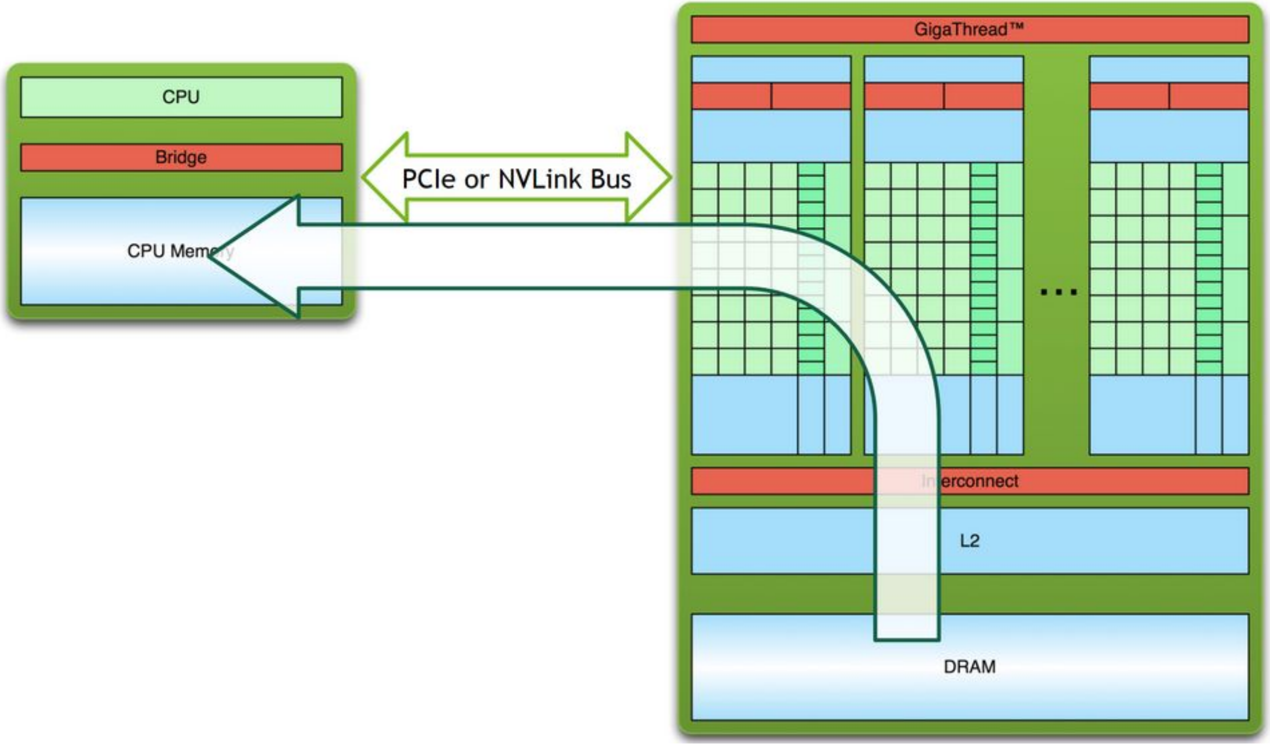


Image source [1]

Memory management

- **The host and device have their own separate memory**:
  - Device pointers point to GPU memory
  - Host pointers point to CPU memory
- **CUDA kernels operate out of device memory**
- CUDA provides functions to **allocate device memory**, **release device memory**, and **transfer data between the host memory and device memory** :

```
cudaMalloc(&ptr, size_in_bytes_to_allocate)     cudaFree(ptr)

cudaMemcpy(destination_ptr,source_ptr, size_in_bytes, direction)
```

Memory management

- **Host pointers :**
  - Typically not passed to device code
  - Typically not dereferenced in device code
- **Device pointers :**
  - Typically passed to device code
  - Typically not dereferenced in host code

For transfers between host and device memory the direction can be :

- Copying data from CPU to GPU
- Copying data from GPU to CPU

```
int* a;
int* d_a;
// Host copy of variable a
a = (int*) malloc(sizeof(int));
// Device copy of variable a
cudaMalloc(&d_a, sizeof(int));
// Set the host value of a
*a = 1;
// Copy the value of a to the device
cudaMemcpy(d_a, a, sizeof(int), cudaMemcpyHostToDevice);
// Launch the kernel to set the value
do_something<<<1,1>>>(d_a);
cudaDeviceSynchronize();
// Copy the value of a back to the host
cudaMemcpy(a, d_a, sizeof(int), cudaMemcpyDeviceToHost);
// Free the allocated memory
free(a);
cudaFree(d_a);
```

Let's take a look at the syntax of cudamalloc

Memory management

- **Host pointers :**
  - Typically not passed to device code
  - Typically not dereferenced in device code
- **Device pointers :**
  - Typically passed to device code
  - Typically not dereferenced in host code

For transfers between host and device memory the direction can be :

- Copying data from CPU to GPU
- Copying data from GPU to CPU

```
int* a;
int* d_a;
// Host copy of variable a
a = (int*) malloc(sizeof(int));
// Device copy of variable a
cudaMalloc(&d_a, sizeof(int));
// Set the host value of a
*a = 1;
// Copy the value of a to the device
cudaMemcpy(d_a, a, sizeof(int), cudaMemcpyHostToDevice);
// Launch the kernel to set the value
do_something<<<1,1>>>(d_a);
cudaDeviceSynchronize();
// Copy the value of a back to the host
cudaMemcpy(a, d_a, sizeof(int), cudaMemcpyDeviceToHost);
// Free the allocated memory
free(a);
cudaFree(d_a);
```

Let's take a look at the syntax of cudamalloc

Remember the order for copying variables from host ⟷ device!

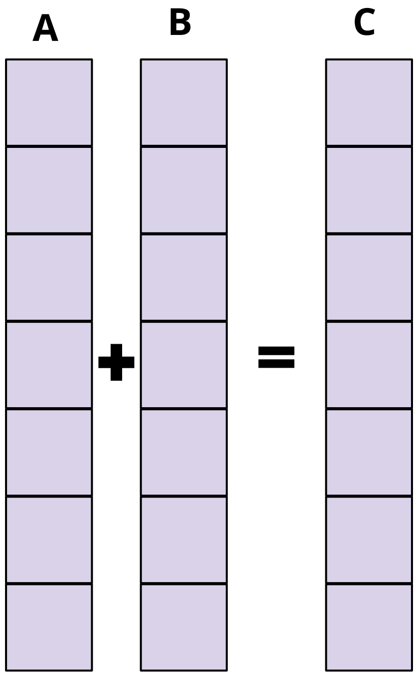
Practical example : Adding two vectors

Lets first start by writing our CUDA kernel :

- __global__ function declaration
- Must return void

Our CUDA kernel now has several arguments e.g. vectors A and B, the resulting vector C and the vector size

```
__global__ void vector_addition(const float *A, const float *B, float *C, int v_size) {

   int idx = threadIdx.x + blockDim.x * blockIdx.x;
   if (idx < v_size)
       C[idx] = A[idx] + B[idx];
}
```



Practical example : Adding two vectors

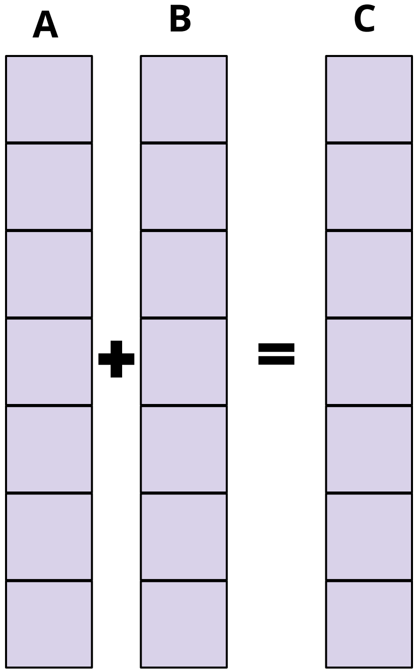
Lets first start by writing our CUDA kernel :

- __global__ function declaration
- Must return void

Our CUDA kernel now has several arguments e.g. vectors A and B, the resulting vector C and the vector size

```
__global__ void vector_addition(const float *A, const float *B, float *C, int v_size) {

   int idx = threadIdx.x + blockDim.x * blockIdx.x;
   if (idx < v_size)
       C[idx] = A[idx] + B[idx];
}
```

We express the vector index in terms of thread and block ID



Practical example : Adding two vectors

Lets first start by writing our CUDA kernel :

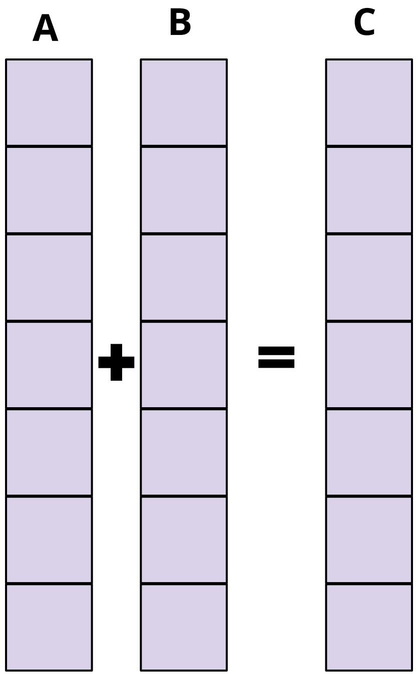
- __global__ function declaration
- Must return void

Our CUDA kernel now has several arguments e.g. vectors A and B, the resulting vector C and the vector size

```
__global__ void vector_addition(const float *A, const float *B, float *C, int v_size) {

   int idx = threadIdx.x + blockDim.x * blockIdx.x;
   if (idx < v_size)
       C[idx] = A[idx] + B[idx];
}
```

We also want to make sure that we don't go beyond our vector range

We express the vector index in terms of thread and block ID

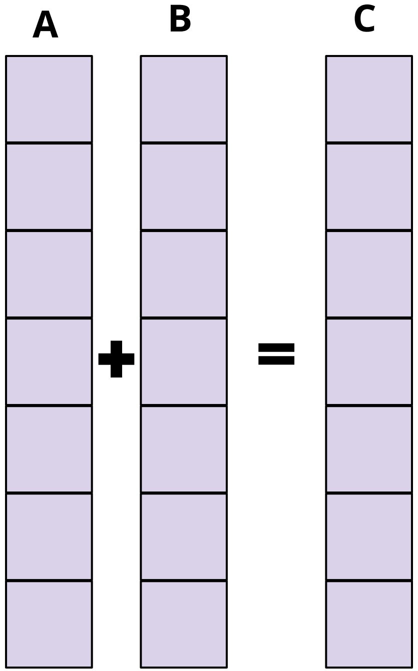


Practical example : Adding two vectors

```
float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
h_A = new float[DSIZE];
h_B = new float[DSIZE];
h_C = new float[DSIZE];

for (int i = 0; i < DSIZE; i++) {
    h_A[i] = rand()/(float)RAND_MAX;
    h_B[i] = rand()/(float)RAND_MAX;
    h_C[i] = 0;
}

cudaMalloc(&d_A, DSIZE*sizeof(float));
cudaMalloc(&d_B, DSIZE*sizeof(float));
cudaMalloc(&d_C, DSIZE*sizeof(float));

cudaMemcpy(d_A, h_A, DSIZE*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, DSIZE*sizeof(float), cudaMemcpyHostToDevice);

vector_addition<<<grid_size, block_size>>>(d_A, d_B, d_C, DSIZE);

cudaMemcpy(h_C, d_C, DSIZE*sizeof(float), cudaMemcpyDeviceToHost);

free(h_A);
free(h_B);
free(h_C);

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Let's start writing our main function!

- We create the necessary host and device pointers

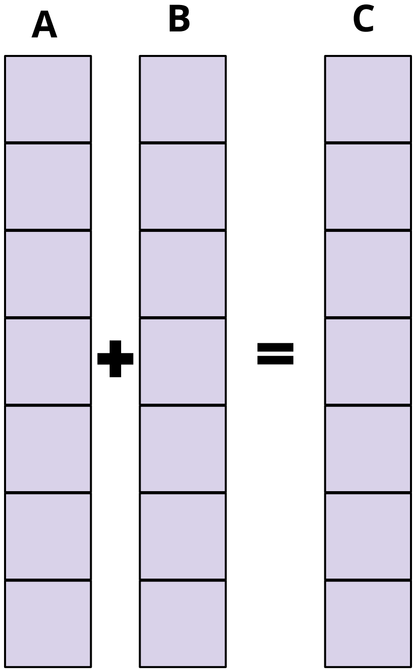


Practical example : Adding two vectors

```
float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
h_A = new float[DSIZE];
h_B = new float[DSIZE];
h_C = new float[DSIZE];

for (int i = 0; i < DSIZE; i++) {
    h_A[i] = rand()/(float)RAND_MAX;
    h_B[i] = rand()/(float)RAND_MAX;
    h_C[i] = 0;
}

cudaMalloc(&d_A, DSIZE*sizeof(float));
cudaMalloc(&d_B, DSIZE*sizeof(float));
cudaMalloc(&d_C, DSIZE*sizeof(float));

cudaMemcpy(d_A, h_A, DSIZE*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, DSIZE*sizeof(float), cudaMemcpyHostToDevice);

vector_addition<<<grid_size, block_size>>>(d_A, d_B, d_C, DSIZE);

cudaMemcpy(h_C, d_C, DSIZE*sizeof(float), cudaMemcpyDeviceToHost);

free(h_A);
free(h_B);
free(h_C);

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Let's start writing our main function!

- We create the necessary host and device pointers
- Allocate the host pointer memory and fill the vectors

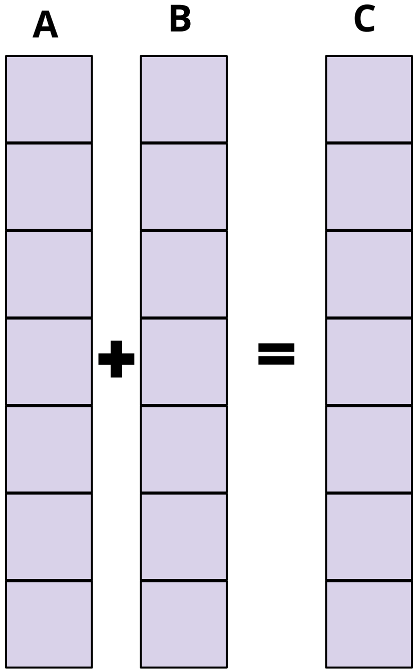


Practical example : Adding two vectors

```
float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
h_A = new float[DSIZE];
h_B = new float[DSIZE];
h_C = new float[DSIZE];

for (int i = 0; i < DSIZE; i++) {
    h_A[i] = rand()/(float)RAND_MAX;
    h_B[i] = rand()/(float)RAND_MAX;
    h_C[i] = 0;
}

cudaMalloc(&d_A, DSIZE*sizeof(float));
cudaMalloc(&d_B, DSIZE*sizeof(float));
cudaMalloc(&d_C, DSIZE*sizeof(float));

cudaMemcpy(d_A, h_A, DSIZE*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, DSIZE*sizeof(float), cudaMemcpyHostToDevice);

vector_addition<<<grid_size, block_size>>>(d_A, d_B, d_C, DSIZE);

cudaMemcpy(h_C, d_C, DSIZE*sizeof(float), cudaMemcpyDeviceToHost);

free(h_A);
free(h_B);
free(h_C);

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Let's start writing our main function!

- We create the necessary host and device pointers
- Allocate the host pointer memory and fill the vectors
- Allocate the necessary memory for the device pointers as well

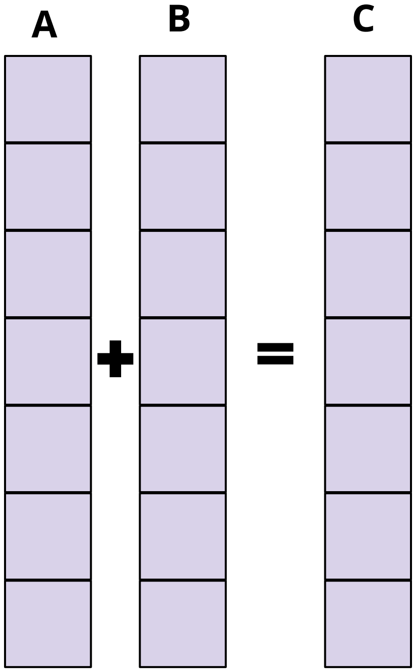


Practical example : Adding two vectors

```
float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
h_A = new float[DSIZE];
h_B = new float[DSIZE];
h_C = new float[DSIZE];

for (int i = 0; i < DSIZE; i++) {
    h_A[i] = rand()/(float)RAND_MAX;
    h_B[i] = rand()/(float)RAND_MAX;
    h_C[i] = 0;
}

cudaMalloc(&d_A, DSIZE*sizeof(float));
cudaMalloc(&d_B, DSIZE*sizeof(float));
cudaMalloc(&d_C, DSIZE*sizeof(float));

cudaMemcpy(d_A, h_A, DSIZE*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, DSIZE*sizeof(float), cudaMemcpyHostToDevice);

vector_addition<<<grid_size, block_size>>>(d_A, d_B, d_C, DSIZE);

cudaMemcpy(h_C, d_C, DSIZE*sizeof(float), cudaMemcpyDeviceToHost);

free(h_A);
free(h_B);
free(h_C);

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Let's start writing our main function!

- We create the necessary host and device pointers
- Allocate the host pointer memory and fill the vectors
- Allocate the necessary memory for the device pointers as well
- Copy data from host to device



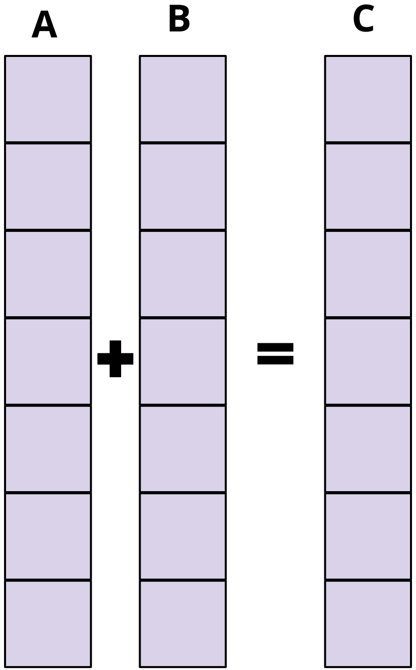
Practical example : Adding two vectors

```
float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
h_A = new float[DSIZE];
h_B = new float[DSIZE];
h_C = new float[DSIZE];

for (int i = 0; i < DSIZE; i++) {
    h_A[i] = rand()/(float)RAND_MAX;
    h_B[i] = rand()/(float)RAND_MAX;
    h_C[i] = 0;
}

cudaMalloc(&d_A, DSIZE*sizeof(float));
cudaMalloc(&d_B, DSIZE*sizeof(float));
cudaMalloc(&d_C, DSIZE*sizeof(float));

cudaMemcpy(d_A, h_A, DSIZE*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, DSIZE*sizeof(float), cudaMemcpyHostToDevice);

vector_addition<<<grid_size, block_size>>>(d_A, d_B, d_C, DSIZE);

cudaMemcpy(h_C, d_C, DSIZE*sizeof(float), cudaMemcpyDeviceToHost);

free(h_A);
free(h_B);
free(h_C);

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Let's start writing our main function!

- We create the necessary host and device pointers
- Allocate the host pointer memory and fill the vectors
- Allocate the necessary memory for the device pointers as well
- Copy data from host to device

- Launch the CUDA kernel



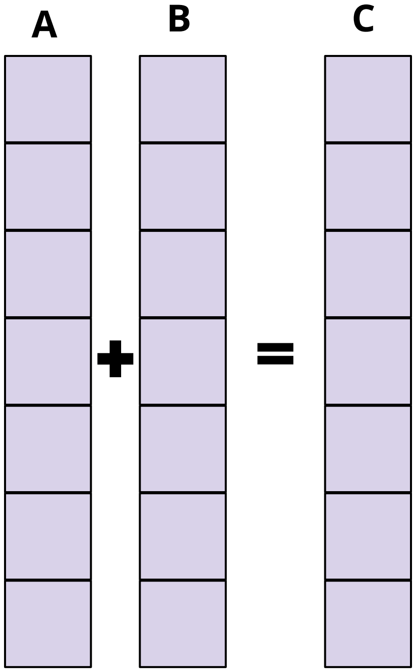
Practical example : Adding two vectors

```
float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
h_A = new float[DSIZE];
h_B = new float[DSIZE];
h_C = new float[DSIZE];

for (int i = 0; i < DSIZE; i++) {
    h_A[i] = rand()/(float)RAND_MAX;
    h_B[i] = rand()/(float)RAND_MAX;
    h_C[i] = 0;
}

cudaMalloc(&d_A, DSIZE*sizeof(float));
cudaMalloc(&d_B, DSIZE*sizeof(float));
cudaMalloc(&d_C, DSIZE*sizeof(float));

cudaMemcpy(d_A, h_A, DSIZE*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, DSIZE*sizeof(float), cudaMemcpyHostToDevice);

vector_addition<<<grid_size, block_size>>>(d_A, d_B, d_C, DSIZE);

cudaMemcpy(h_C, d_C, DSIZE*sizeof(float), cudaMemcpyDeviceToHost);

free(h_A);
free(h_B);
free(h_C);

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Let's start writing our main function!

- We create the necessary host and device pointers
- Allocate the host pointer memory and fill the vectors
- Allocate the necessary memory for the device pointers as well
- Copy data from host to device

- Launch the CUDA kernel
- Copy data from the device back to the host

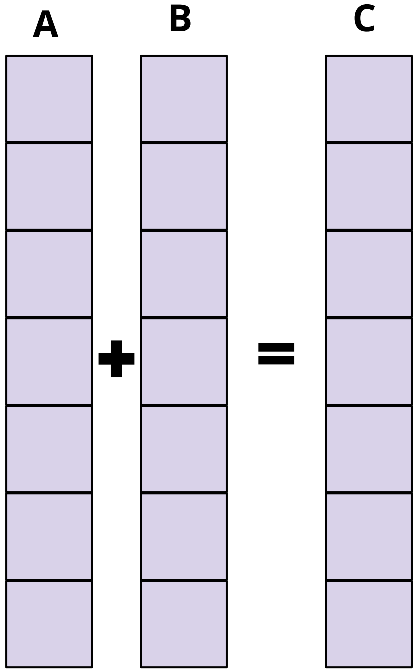


Practical example : Adding two vectors

```
float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
h_A = new float[DSIZE];
h_B = new float[DSIZE];
h_C = new float[DSIZE];

for (int i = 0; i < DSIZE; i++) {
    h_A[i] = rand()/(float)RAND_MAX;
    h_B[i] = rand()/(float)RAND_MAX;
    h_C[i] = 0;
}

cudaMalloc(&d_A, DSIZE*sizeof(float));
cudaMalloc(&d_B, DSIZE*sizeof(float));
cudaMalloc(&d_C, DSIZE*sizeof(float));

cudaMemcpy(d_A, h_A, DSIZE*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, DSIZE*sizeof(float), cudaMemcpyHostToDevice);

vector_addition<<<grid_size, block_size>>>(d_A, d_B, d_C, DSIZE);

cudaMemcpy(h_C, d_C, DSIZE*sizeof(float), cudaMemcpyDeviceToHost);

free(h_A);
free(h_B);
free(h_C);

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Let's start writing our main function!

- We create the necessary host and device pointers
- Allocate the host pointer memory and fill the vectors
- Allocate the necessary memory for the device pointers as well
- Copy data from host to device



- Launch the CUDA kernel
- Copy data from the device back to the host
- Delete the pointers in order to free the host and device memory

Practical example : Adding two vectors

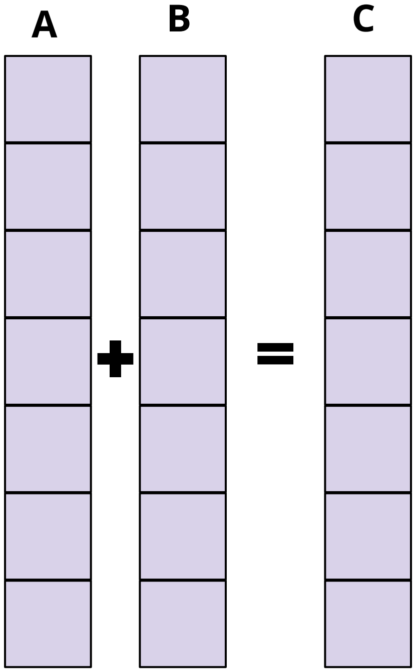
```
float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
h_A = new float[DSIZE];
h_B = new float[DSIZE];
h_C = new float[DSIZE];

for (int i = 0; i < DSIZE; i++) {
    h_A[i] = rand()/(float)RAND_MAX;
    h_B[i] = rand()/(float)RAND_MAX;
    h_C[i] = 0;
}

cudaMalloc(&d_A, DSIZE*sizeof(float));
cudaMalloc(&d_B, DSIZE*sizeof(float));
cudaMalloc(&d_C, DSIZE*sizeof(float));

cudaMemcpy(d_A, h_A, DSIZE*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, DSIZE*sizeof(float), cudaMemcpyHostToDevice);

vector_addition<<<grid_size, block_size>>>(d_A, d_B, d_C, DSIZE);

cudaMemcpy(h_C, d_C, DSIZE*sizeof(float), cudaMemcpyDeviceToHost);

free(h_A);
free(h_B);
free(h_C);

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Let's put this all together!

Exercise

```
ssh <username>@login.hep.wisc.edu
ssh g38nXX
touch vector_addition.cu
# Copy this into the .cu file
export LD_LIBRARY_PATH=/usr/local/cuda/lib
export PATH=$PATH:/usr/local/cuda/bin
nvcc vector_addition.cu -o vector_addition
./vector_addition
```

- Lets try changing the grid/block size.
- How can we ensure that the number of threads is enough ?



Practical example : Adding two vectors

```
float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
h_A = new float[DSIZE];
h_B = new float[DSIZE];
h_C = new float[DSIZE];

for (int i = 0; i < DSIZE; i++) {
     h_A[i] = rand()/(float)RAND_MAX;
     h_B[i] = rand()/(float)RAND_MAX;
     h_C[i] = 0;
}

cudaMalloc(&d_A, DSIZE*sizeof(float));
cudaMalloc(&d_B, DSIZE*sizeof(float));
cudaMalloc(&d_C, DSIZE*sizeof(float));

cudaMemcpy(d_A, h_A, DSIZE*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, DSIZE*sizeof(float), cudaMemcpyHostToDevice);

vector_addition<<<grid_size, block_size>>>(d_A, d_B, d_C, DSIZE);

cudaMemcpy(h_C, d_C, DSIZE*sizeof(float), cudaMemcpyDeviceToHost);

free(h_A);
free(h_B);
free(h_C);

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

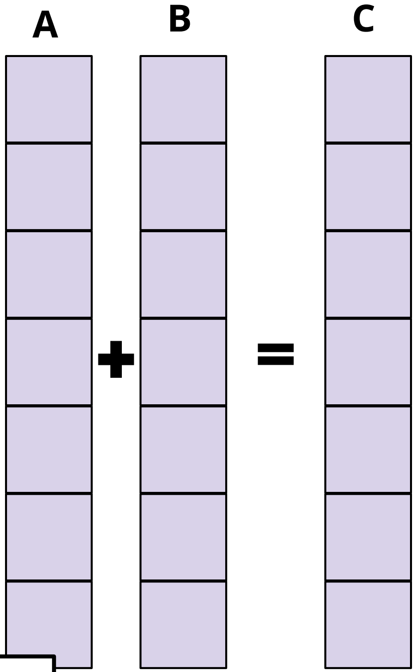
Let's put this all together!

Exercise

```
ssh <username>@login.hep.wisc.edu
ssh g38nXX
touch vector_addition.cu
# Copy this into the .cu file
export LD_LIBRARY_PATH=/usr/local/cuda/lib
export PATH=$PATH:/usr/local/cuda/bin
nvcc vector_addition.cu -o vector_addition
./vector_addition
```

- Lets try changing the grid/block size.

gridSize = (Length of vector + Block size - 1) / Block size



Synchronization

Synchronization

- In the previous lecture we learnt that CUDA kernel calls are asynchronous :
  - Once the kernel is launched the main program that is executed on the CPU continues normally
- Additionally, execution order of blocks on a SMs is arbitrary
  - We need a way to synchronise!

- We saw that call to **CudaDeviceSynchronize()** from host blocks the CPU execution until all work launched on the device has finished.

- Includes both:
  - kernel launches
  - memory copies

```
#include <stdio.h>

__global__ void cuda_hello(){
   printf("Hello World from GPU");
}

int main() {
 int gridDim = 1;
 int blockDim = 1;
 cuda_hello<<<gridDim,blockDim>>>();
 return 0;
}
```

- **Why is nothing printed out on the screen?**
  - Lets try and change the number of threads/block
  - Does this have any impact?

Grid level synchronization

Synchronization

For each kernel launch with N threads/block & M blocks :

- Execution order of threads within one block is arbitrary :
  - Only exception are threads in the same warp which are processed simultaneously
- We might have a problem, where we require all threads in a specific block to have completed execution of a specific task before continuing the next task
- To synchronize threads within one block one can call **__syncthreads()** within the kernel

```
__global__ void myKernel () {
    for (int i = threadIdx.x; i < N; i++) {
        Fill variable[threadIdx.x]
    }
    __syncthreads();
    for (int i = threadIdx.x; i < N; i++) {
        Use variable[threadIdx.x]
    }
}
```

Block level synchronization

Synchronization

For each kernel launch with N threads/block & M blocks :

- Execution order of threads within one block is arbitrary :
  - Only exception are threads in the same warp which are processed simultaneously
- We might have a problem, where we require all threads in a specific block to have completed execution of a specific task before continuing the next task
- To synchronize threads within one block one can call **__syncthreads()** within the kernel

```
__global__ void myKernel () {
    for (int i = threadIdx.x; i < N; i++) {
        Fill variable[threadIdx.x]
    }
    __syncthreads();
    for (int i = threadIdx.x; i < N; i++) {
        Use variable[threadIdx.x]
    }
}
```

Exercise

- Let's try and change a bit the add_vector kernel
- What can we do that would need block level synchronization?

Error handling

Error handling

- Error codes can be converted to a human-readable error messages with the following CUDA run- time function:

```
char* cudaGetErrorString(cudaError_t error)
```

- A common practice is to wrap CUDA calls in utility functions that manage the error returned :

```
int* a;
// Illegal: cannot allocate a negative number of bytes
cudaError_t err = cudaMalloc(&a, -1);
if (err != cudaSuccess) {
    printf("CUDA error %s\n", cudaGetErrorString(err));
    exit(-1);
}
```

- To detect errors in a kernel launch, we can use the API call **cudaGetLastError()** which returns the error code for whatever the last CUDA API call was.

```
cudaError_t err = cudaGetLastError();
```

- For errors that occurs asynchronously during the kernel launch, **cudaDeviceSynchronize()** has to be invoked after the kernel in order to return any errors associated with the kernel launch.

Error handling

```
// error checking macro
#define cudaCheckErrors(msg)                                      \
   do {                                                           \
       cudaError_t __err = cudaGetLastError();                    \
       if (__err != cudaSuccess) {                                \
           fprintf(stderr, "Fatal error: %s (%s at %s:%d)\n",     \
                   msg, cudaGetErrorString(__err),                \
                   __FILE__, __LINE__);                           \
           fprintf(stderr, "*** FAILED - ABORTING\n");            \
           exit(1);                                               \
       }                                                          \
   } while (0)
```

We can define a utility function outside of our main program to help us check for CUDA errors

Lets try this out !

- You can copy this from here into our script.
- Let's add a mistake somewhere
- Let's compile and run our script without error-checking
  - What do you observe?
- Lets add error-checking
  - What happened now?

Compilation

- Compiling a CUDA program is similar to compiling a C/C++ program.
- Cuda code should be typically stored in a file with extension .cu
- NVIDIA provides a CUDA compiler called **nvcc** :
    - nvcc is called for CUDA parts
    - gcc is called for c++ parts
    - nvcc converts .cu files into C++ for the host system and CUDA assembly or binary instructions for the device
- Usage :

```
nvcc myCudaProgram.cu -o myCudaProgram
```

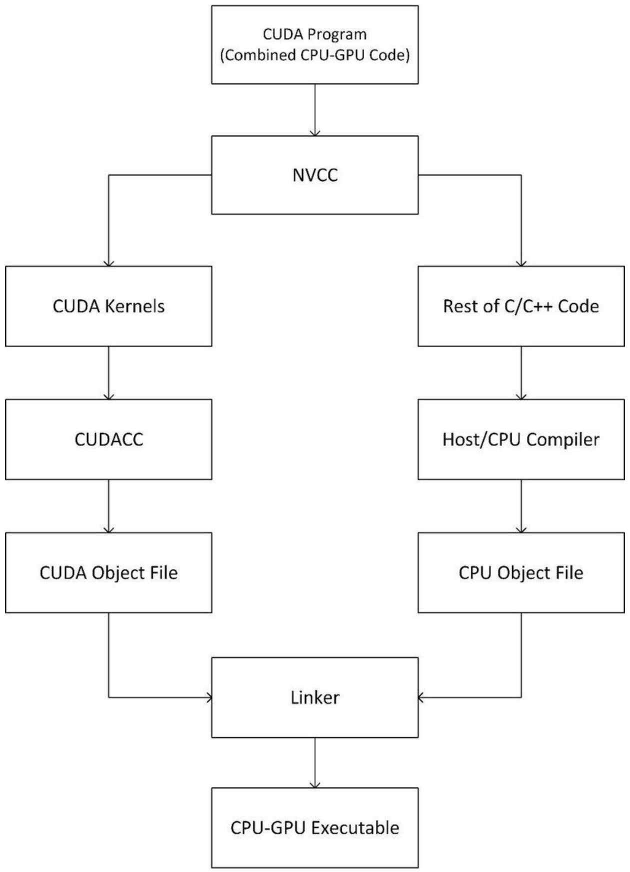


Image source [i]

Wrapping-up

Overview of today's lecture

- We learnt how to copy data to and from the host and the device
  - We wrote our first CUDA program that adds two vectors!
- We discussed the different levels of synchronization
  - Block level & grid level
- Error handling :
  - We learnt how to check for errors in our GPU programm

Assignment for next week

- Assignment can be found here (**Week 3**) :

  https://github.com/ckoraka/tac-hep-gpus

- To clone :
  - git clone git@github.com:ckoraka/tac-hep-gpus.git
- **Due Friday October 4th**
- Please upload assignment here :
  - https://pages.hep.wisc.edu/~ckoraka/assignments/TAC-HEP/
  - Upload only 1 .pdf file with all exercises
  - If you also have your code on git, please add the link to your repository in the pdf file you upload.

Next week

We will dive deeper into CUDA

- Optimizing the number of threads and blocks
- Synchronization at grid and block level
- Memory access patterns and coalesced memory accesses
- Static and dynamic shared memory
- Optimizing memory performance
- Race conditions and atomic operations
- The default CUDA stream

Back-up

Resources

1. NVIDIA Deep Learning Institute material link
2. 10th Thematic CERN School of Computing material link
3. Nvidia turing architecture white paper link
4. CUDA programming guide link
5. CUDA runtime API documentation link
6. CUDA profiler user's guide link