

Introduction to HIP Programming

Tom Papatheodore

HPC Engineer

Oak Ridge Leadership Computing Facility

System Acceptance & User Environment



ORNL is managed by UT-Battelle LLC for the US Department of Energy



U.S. DEPARTMENT OF
ENERGY

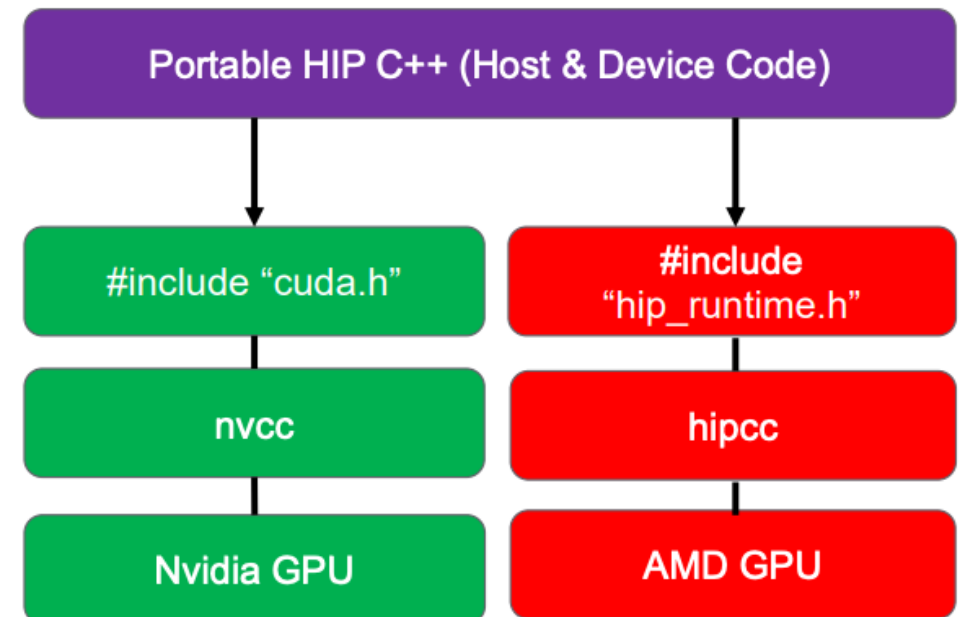
Outline

- What is HIP?
- Heterogeneous programming (CPU + GPU)
- Structure of a basic HIP program
- HIP error checking
- Timing HIP codes
- Multi-D HIP grids
- Hands-on session
- Optimization example
- Hands-on session

What is HIP?

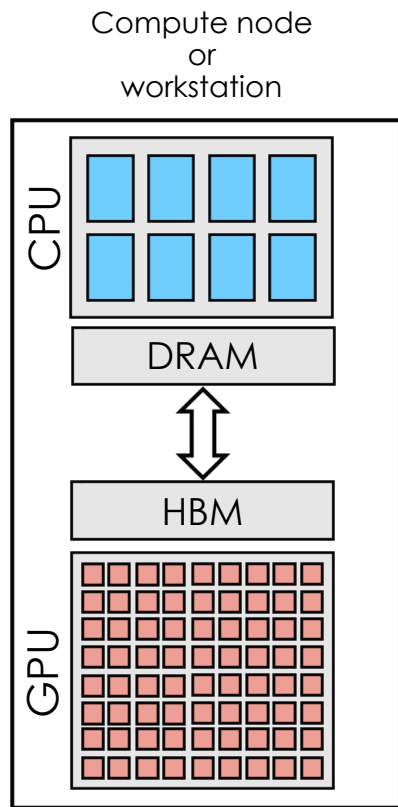
Heterogeneous-Compute Interface for Portability (HIP)

- C++ runtime API and kernel language that allows developers to create portable applications that can run on AMD and NVIDIA GPUs.
- Syntactically similar to CUDA so that most API calls can be converted from CUDA to HIP with a simple `cuda` → `hip` translation.

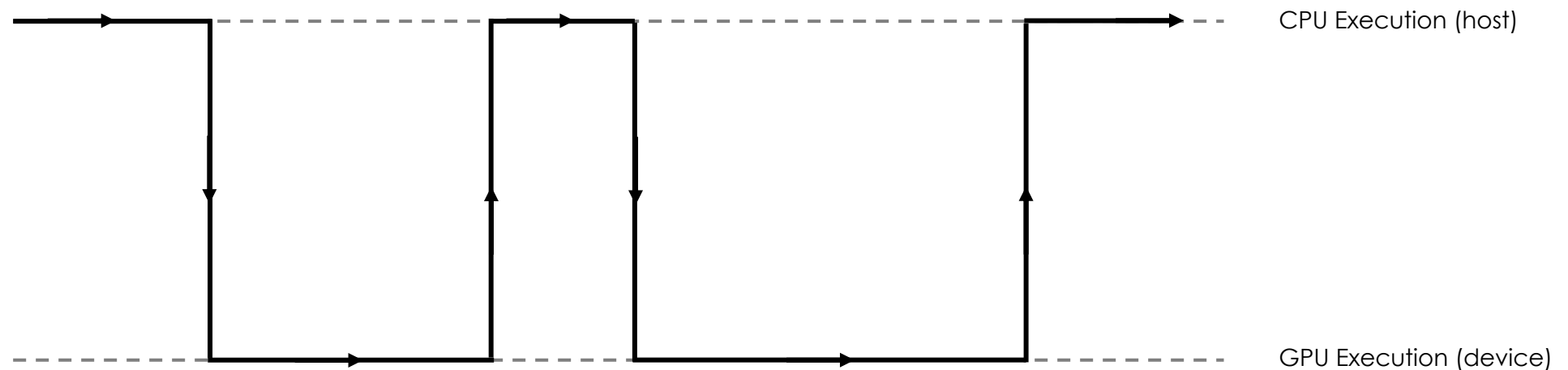


Heterogeneous Programming Model

- CPU and GPU have their own physical memories
- CPU has several cores, low latency, and lower memory bandwidth than GPU
- GPU has many cores, high throughput, and higher memory bandwidth than CPU



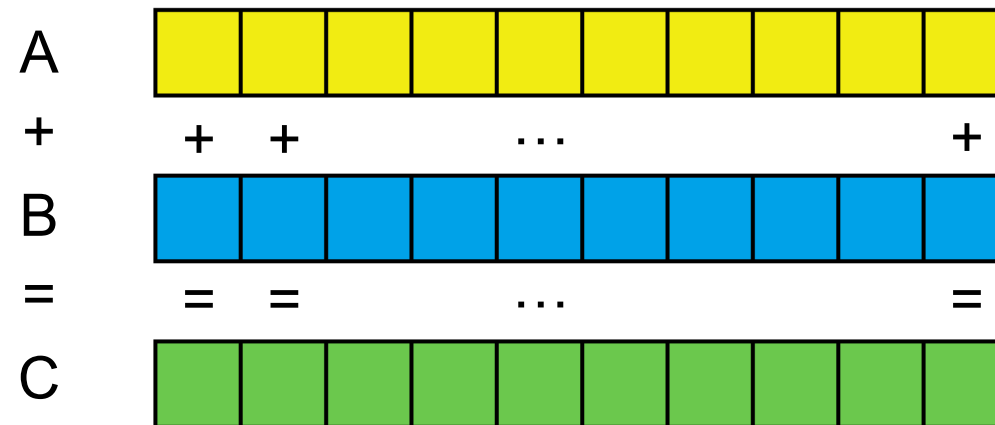
Co-processor model: As an application runs, execution is passed back and forth between the CPU and GPU



NOTE: data transfers can be costly!

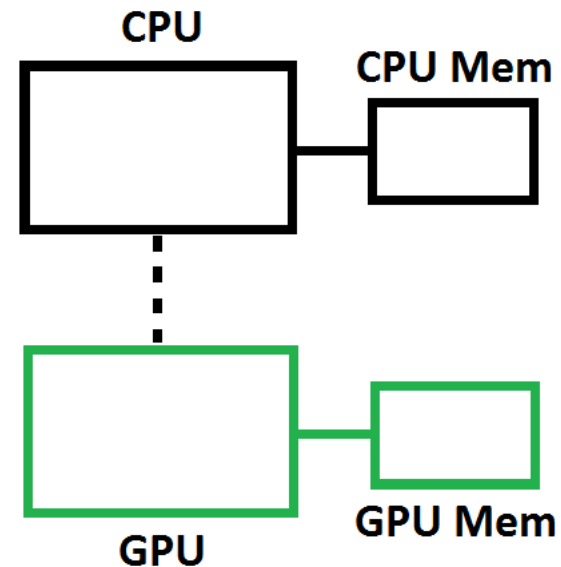
Example: Vector Addition

- Embarrassingly Parallel; each element-wise addition is completely independent from all others, so all elements can be computed at the same time.
- Let's see how this can be parallelized on a GPU using HIP...



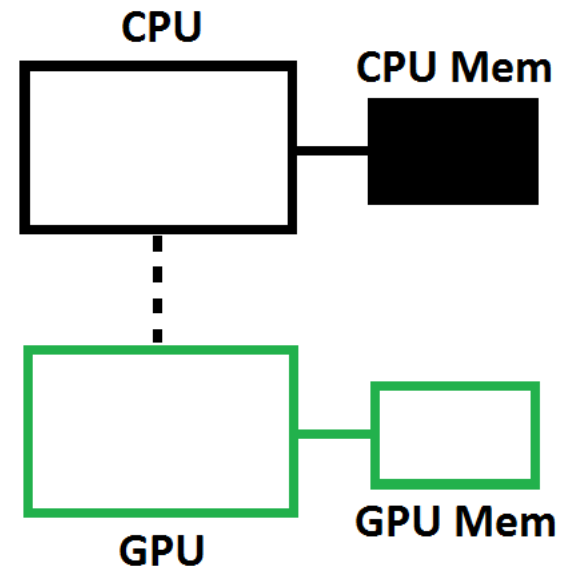
Example: Vector Addition Program Outline

```
int main(){  
  
    // Allocate memory for array on host  
  
    // Allocate memory for array on device  
  
    // Fill array on host  
  
    // Copy data from host array to device array  
  
    // Do something on device (e.g. vector addition)  
  
    // Copy data from device array to host array  
  
    // Check data for correctness  
  
    // Free Host Memory  
  
    // Free Device Memory  
  
}
```



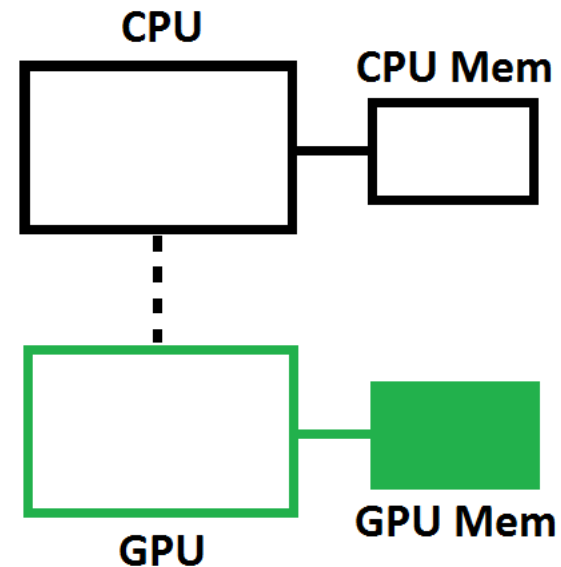
Example: Vector Addition Program Outline

```
int main(){  
  
    // Allocate memory for array on host  
  
    // Allocate memory for array on device  
  
    // Fill array on host  
  
    // Copy data from host array to device array  
  
    // Do something on device (e.g. vector addition)  
  
    // Copy data from device array to host array  
  
    // Check data for correctness  
  
    // Free Host Memory  
  
    // Free Device Memory  
  
}
```



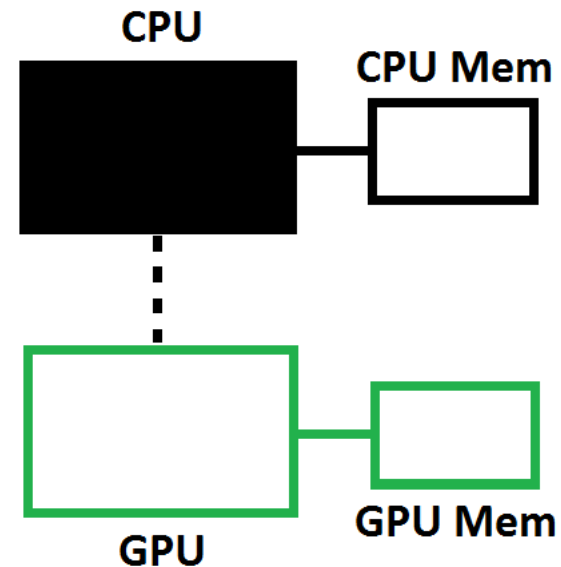
Example: Vector Addition Program Outline

```
int main(){  
  
    // Allocate memory for array on host  
  
    // Allocate memory for array on device  
  
    // Fill array on host  
  
    // Copy data from host array to device array  
  
    // Do something on device (e.g. vector addition)  
  
    // Copy data from device array to host array  
  
    // Check data for correctness  
  
    // Free Host Memory  
  
    // Free Device Memory  
  
}
```



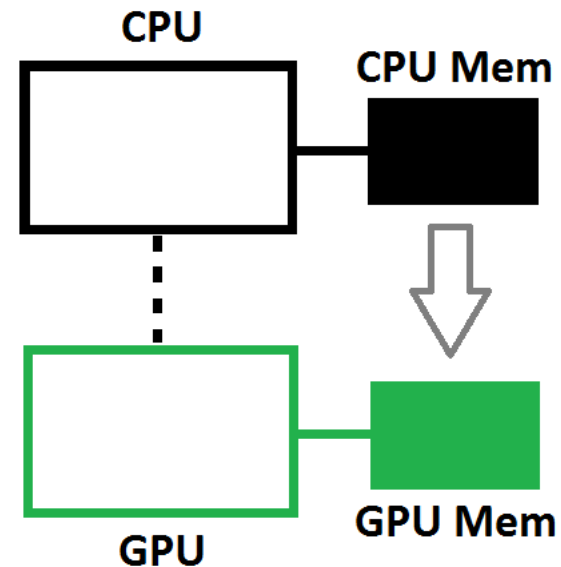
Example: Vector Addition Program Outline

```
int main(){  
  
    // Allocate memory for array on host  
  
    // Allocate memory for array on device  
  
    // Fill array on host  
  
    // Copy data from host array to device array  
  
    // Do something on device (e.g. vector addition)  
  
    // Copy data from device array to host array  
  
    // Check data for correctness  
  
    // Free Host Memory  
  
    // Free Device Memory  
  
}
```



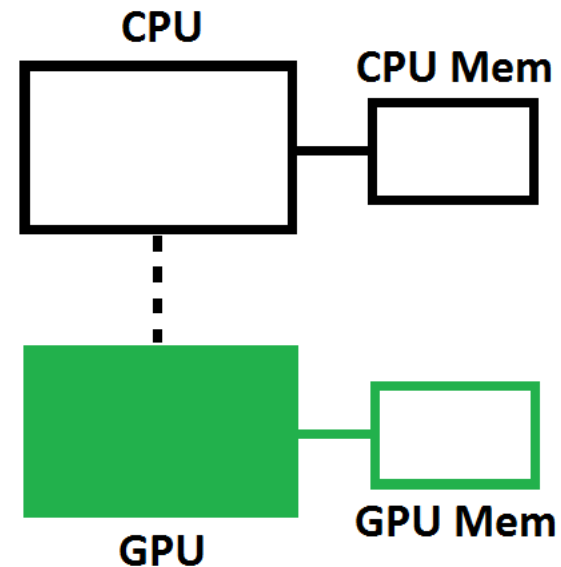
Example: Vector Addition Program Outline

```
int main(){  
  
    // Allocate memory for array on host  
  
    // Allocate memory for array on device  
  
    // Fill array on host  
  
    // Copy data from host array to device array  
  
    // Do something on device (e.g. vector addition)  
  
    // Copy data from device array to host array  
  
    // Check data for correctness  
  
    // Free Host Memory  
  
    // Free Device Memory  
  
}
```



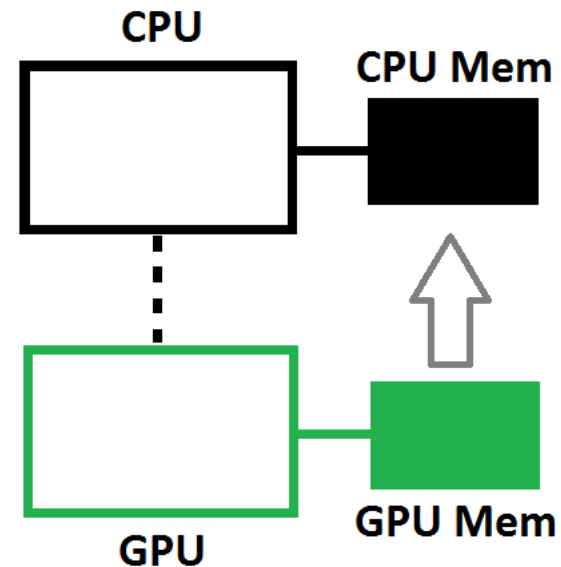
Example: Vector Addition Program Outline

```
int main(){  
  
    // Allocate memory for array on host  
  
    // Allocate memory for array on device  
  
    // Fill array on host  
  
    // Copy data from host array to device array  
  
    // Do something on device (e.g. vector addition)  
  
    // Copy data from device array to host array  
  
    // Check data for correctness  
  
    // Free Host Memory  
  
    // Free Device Memory  
  
}
```



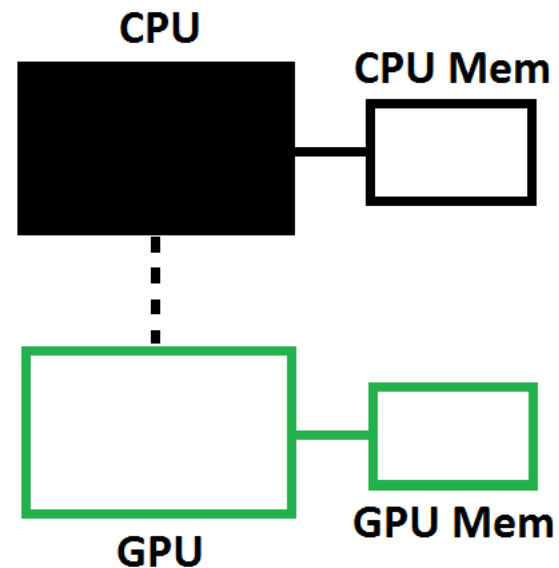
Example: Vector Addition Program Outline

```
int main(){  
  
    // Allocate memory for array on host  
  
    // Allocate memory for array on device  
  
    // Fill array on host  
  
    // Copy data from host array to device array  
  
    // Do something on device (e.g. vector addition)  
  
    // Copy data from device array to host array  
  
    // Check data for correctness  
  
    // Free Host Memory  
  
    // Free Device Memory  
  
}
```



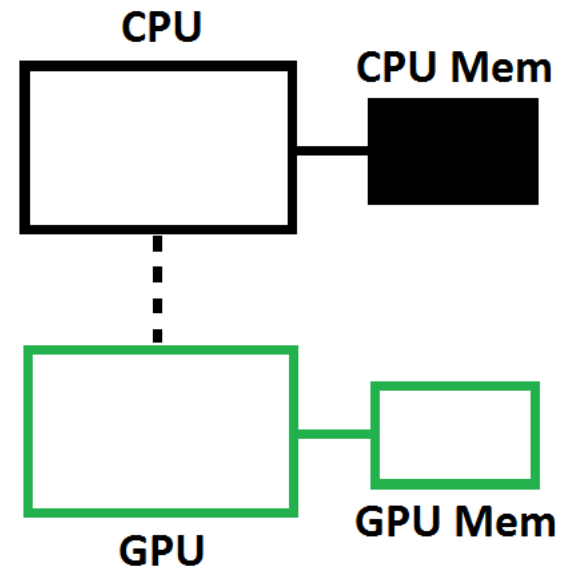
Example: Vector Addition Program Outline

```
int main(){  
  
    // Allocate memory for array on host  
  
    // Allocate memory for array on device  
  
    // Fill array on host  
  
    // Copy data from host array to device array  
  
    // Do something on device (e.g. vector addition)  
  
    // Copy data from device array to host array  
  
    // Check data for correctness  
  
    // Free Host Memory  
  
    // Free Device Memory  
  
}
```



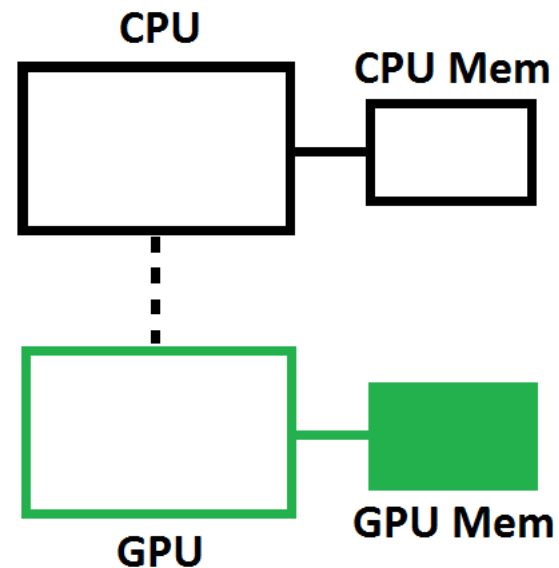
Example: Vector Addition Program Outline

```
int main(){  
  
    // Allocate memory for array on host  
  
    // Allocate memory for array on device  
  
    // Fill array on host  
  
    // Copy data from host array to device array  
  
    // Do something on device (e.g. vector addition)  
  
    // Copy data from device array to host array  
  
    // Check data for correctness  
  
    // Free Host Memory  
  
    // Free Device Memory  
  
}
```



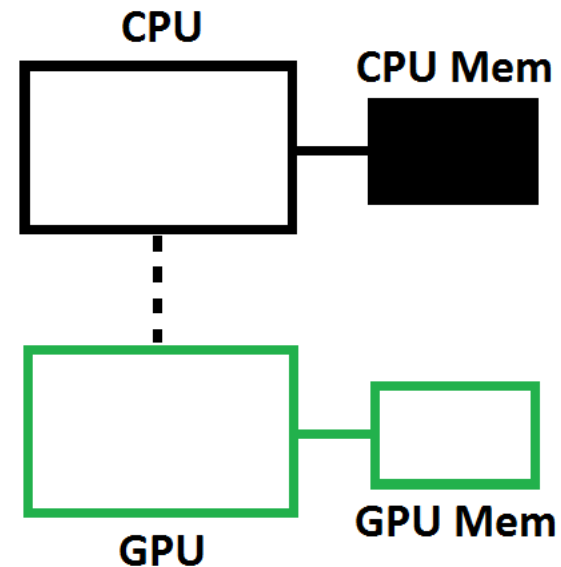
Example: Vector Addition Program Outline

```
int main(){  
  
    // Allocate memory for array on host  
  
    // Allocate memory for array on device  
  
    // Fill array on host  
  
    // Copy data from host array to device array  
  
    // Do something on device (e.g. vector addition)  
  
    // Copy data from device array to host array  
  
    // Check data for correctness  
  
    // Free Host Memory  
  
    // Free Device Memory  
  
}
```



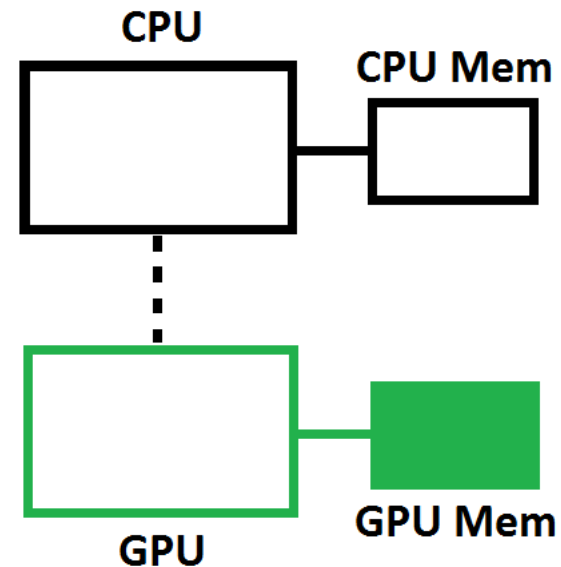
Example: Vector Addition Program Outline (HIP)

```
int main(){  
    . . .  
    // Allocate memory for array on host  
    size_t bytes = N*sizeof(double);  
    double *A = (double*)malloc(bytes);  
    double *B = (double*)malloc(bytes);  
    double *C = (double*)malloc(bytes);  
    . . .  
}
```



Example: Vector Addition Program Outline (HIP)

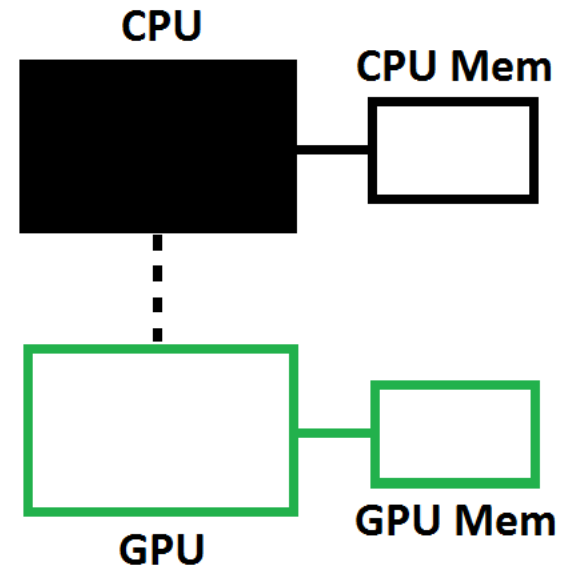
```
int main(){  
    . . .  
    // Allocate memory for array on device  
    double *d_A, *d_B, *d_C;  
    hipMalloc(&d_A, bytes);  
    hipMalloc(&d_B, bytes);  
    hipMalloc(&d_C, bytes);  
    . . .  
}
```



```
hipError_t hipMalloc( void** devPtr, size_t size )
```

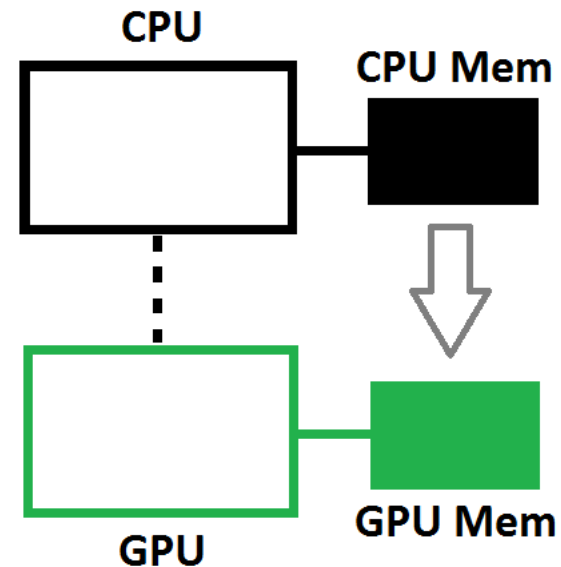
Example: Vector Addition Program Outline (HIP)

```
int main(){  
    . . .  
    // Fill array on host  
    for(int i=0; i<N; i++)  
    {  
        A[i] = 1.0;  
        B[i] = 2.0;  
        C[i] = 0.0;  
    }  
    . . .  
}
```



Example: Vector Addition Program Outline (HIP)

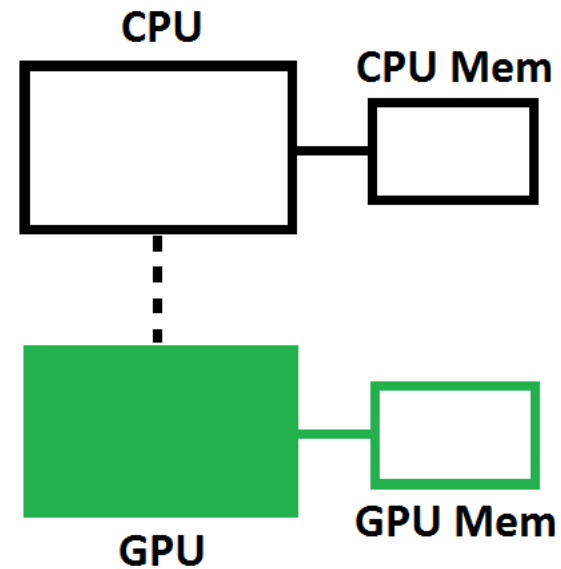
```
int main(){  
    . . .  
    // Copy data from host array to device array  
    hipMemcpy(d_A, A, bytes, hipMemcpyHostToDevice);  
    hipMemcpy(d_B, B, bytes, hipMemcpyHostToDevice);  
    . . .  
}
```



```
hipError_t hipMemcpy( void* dst, const void* src, size_t count, hipMemcpyKind kind )
```

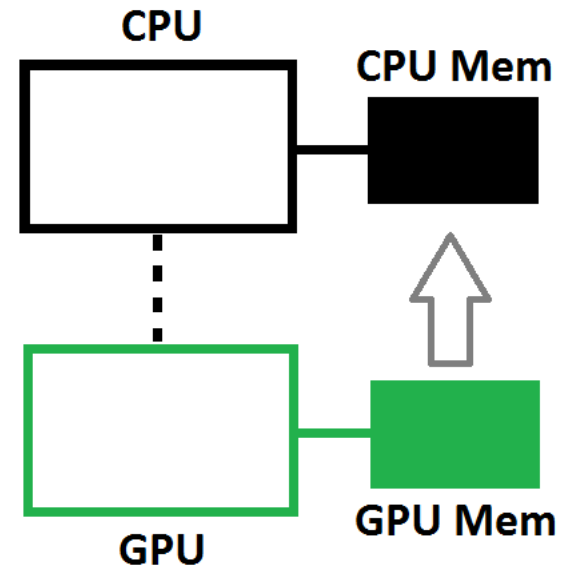
Example: Vector Addition Program Outline (HIP)

```
int main(){  
    . . .  
    // Do something on device (e.g. vector addition)  
    // We'll come back to this soon  
    . . .  
}
```



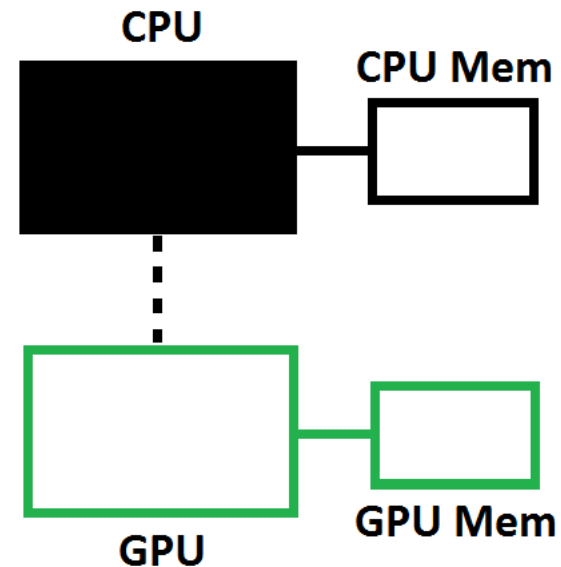
Example: Vector Addition Program Outline (HIP)

```
int main(){  
    . . .  
    // Copy data from device array to host array  
    hipMemcpy(C, d_C, bytes, hipMemcpyDeviceToHost);  
    . . .  
}
```



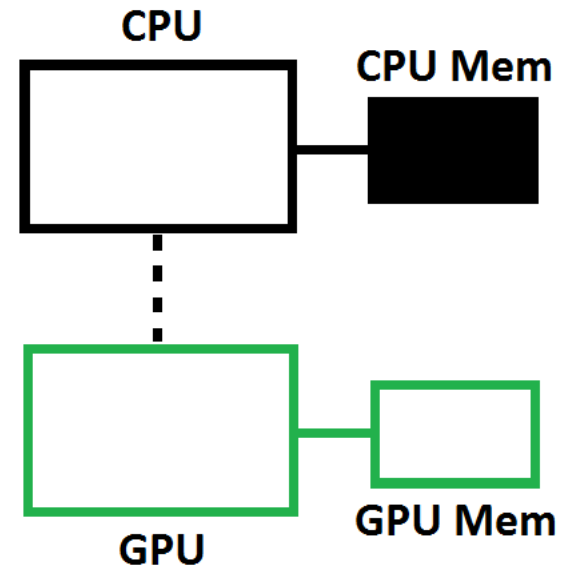
Example: Vector Addition Program Outline (HIP)

```
int main(){  
    . . .  
    // Check data for correctness  
    double tolerance = 1.0e-14;  
    for(int i=0; i<N; i++)  
    {  
        if( fabs(C[i] - 3.0) > tolerance )  
        {  
            printf("Error: value of C[%d] = %f instead of 3.0\n", i, C[i]);  
            exit(-1);  
        }  
    }  
    . . .  
}
```



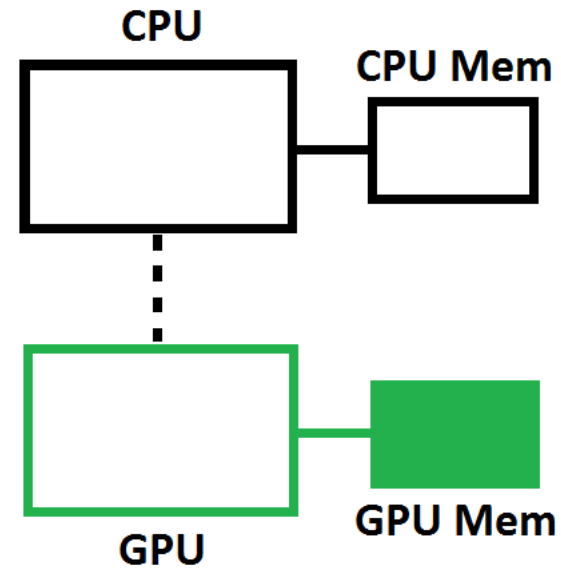
Example: Vector Addition Program Outline (HIP)

```
int main() {  
    . . .  
    // Free Host Memory  
    free(A);  
    free(B);  
    free(C);  
    . . .  
}
```



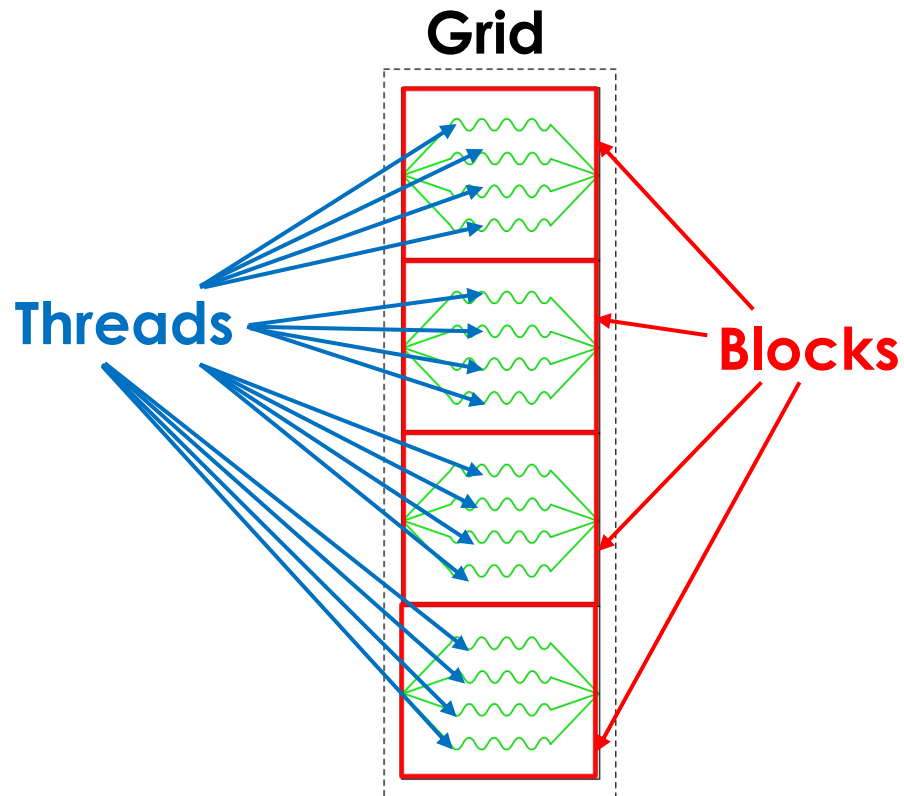
Example: Vector Addition Program Outline (HIP)

```
int main() {  
    . . .  
    // Free Device Memory  
    hipFree(d_A);  
    hipFree(d_B);  
    hipFree(d_C);  
    . . .  
}
```



```
hipError_t hipFree( void* devPtr )
```


HIP Thread, Block, Grid Hierarchy



A grid of threads is spawned, where the threads are partitioned into blocks.

- Threads within a block can cooperate when performing calculations.

```
N = 16;  
thr_per_blk = 4;  
blk_in_grid = ceil(float(N) / thr_per_blk);
```

Example: Vector Addition (HIP Kernel)

A **kernel** in HIP programming is a function that runs on the GPU.

- But how is it different than a normal function?

Serial function

```
void vector_addition(double *a, double *b, double *c){  
    for (int i=0; i<N, i++){  
        c[i]= a[i] + b[i];  
    }  
}
```

A single process iterates through the loop and adds the vectors element-by-element (sequentially).

GPU kernel

```
__global__ void vector_addition(double *a, double *b, double *c)  
{  
    int id = blockDim.x * blockIdx.x + threadIdx.x;  
    if (id < N) c[id] = a[id] + b[id];  
}
```

All GPU threads run same kernel function, but each thread is assigned a unique global ID to know which element(s) to calculate.

Example: Vector Addition (HIP Kernel)

GPU kernel

```
__global__ void vector_addition(double *a, double *b, double *c)
{
    int id = blockDim.x * blockIdx.x + threadIdx.x;

    if (id < N) c[id] = a[id] + b[id];
}
```

`__global__`

Indicates the function is a HIP kernel function – called by the host (CPU) and executed on the device (GPU).

Example: Vector Addition (HIP Kernel)

GPU kernel

```
__global__ void vector_addition(double *a, double *b, double *c)
{
    int id = blockDim.x * blockIdx.x + threadIdx.x;

    if (id < N) c[id] = a[id] + b[id];
}
```

__void__

Indicates the kernel does not return anything.

Example: Vector Addition (HIP Kernel)

GPU kernel

```
__global__ void vector_addition(double *a, double *b, double *c)
{
    int id = blockDim.x * blockIdx.x + threadIdx.x;

    if (id < N) c[id] = a[id] + b[id];
}
```

double *a, double *b, double *c

Kernel function arguments.

- a, b, c are pointers to device memory (allocated with `hipMalloc`)

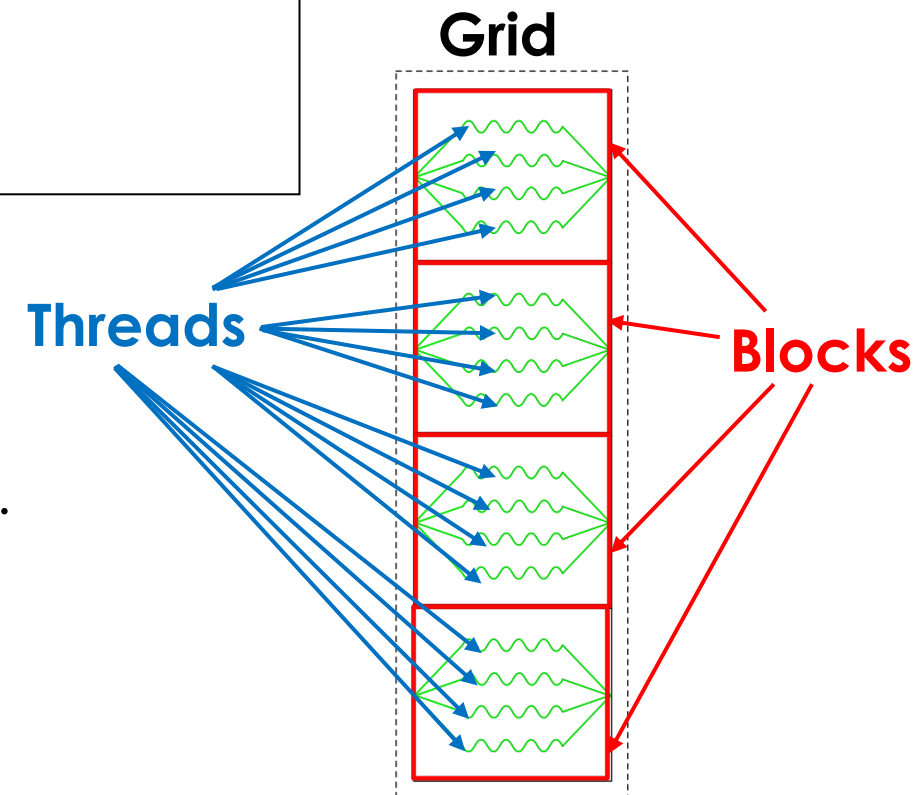
Example: Vector Addition (HIP Kernel)

GPU kernel

```
__global__ void vector_addition(double *a, double *b, double *c)
{
    int id = blockDim.x * blockIdx.x + threadIdx.x;

    if (id < N) c[id] = a[id] + b[id];
}
```

`int id = blockDim.x * blockIdx.x + threadIdx.x;`
This defines a unique thread ID among all threads in a grid.



Example: Vector Addition (HIP Kernel)

GPU kernel

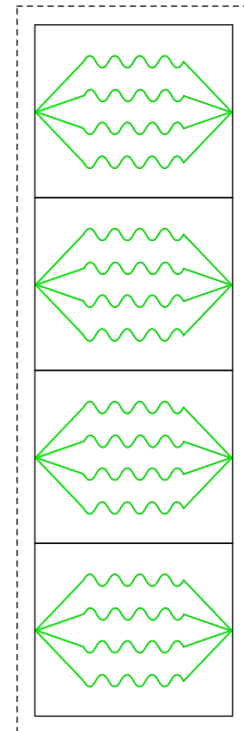
```
__global__ void vector_addition(double *a, double *b, double *c)
{
    4
    int id = blockDim.x * blockIdx.x + threadIdx.x;

    if (id < N) c[id] = a[id] + b[id];
}
```

blockDim

Gives the number of threads within each block (x-dimension for 1D).

- E.g., 4 threads per block



Example: Vector Addition (HIP Kernel)

GPU kernel

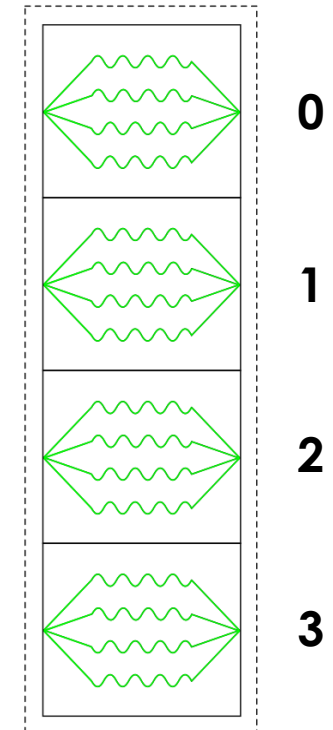
```
__global__ void vector_addition(double *a, double *b, double *c)
{
    4           0-3
    int id = blockDim.x * blockIdx.x + threadIdx.x;

    if (id < N) c[id] = a[id] + b[id];
}
```

blockIdx

Specifies the block index of the thread (within the grid of blocks).

- I.e., which block the thread is in



Example: Vector Addition (HIP Kernel)

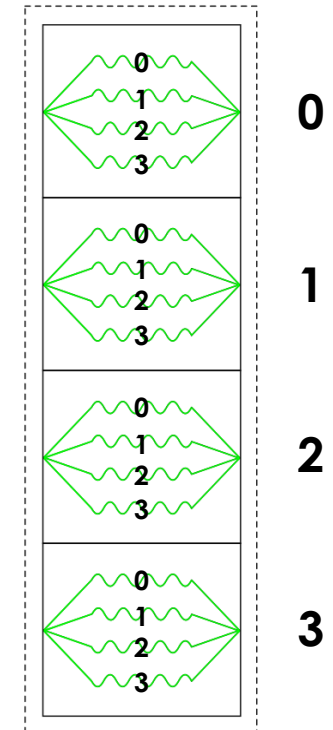
GPU kernel

```
__global__ void vector_addition(double *a, double *b, double *c)
{
    4           0-3           0-3
    int id = blockDim.x * blockIdx.x + threadIdx.x;

    if (id < N) c[id] = a[id] + b[id];
}
```

threadIdx

Specifies a thread's local ID within a thread block.



Example: Vector Addition (HIP Kernel)

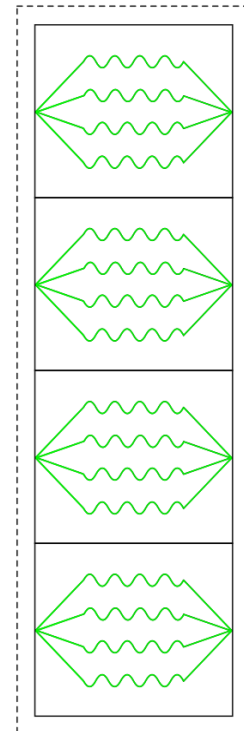
GPU kernel

```
__global__ void vector_addition(double *a, double *b, double *c)
{
    int id = blockDim.x * blockIdx.x + threadIdx.x;

    if (id < N) c[id] = a[id] + b[id];
}
```

`int id = blockDim.x * blockIdx.x + threadIdx.x;`

This defines a unique thread ID among all threads in a grid.



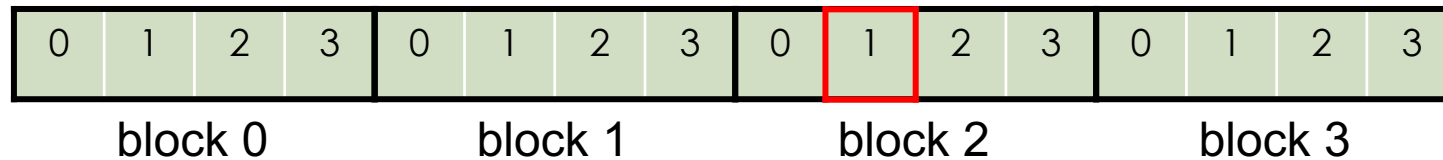
Example: Vector Addition (HIP Kernel)

GPU kernel

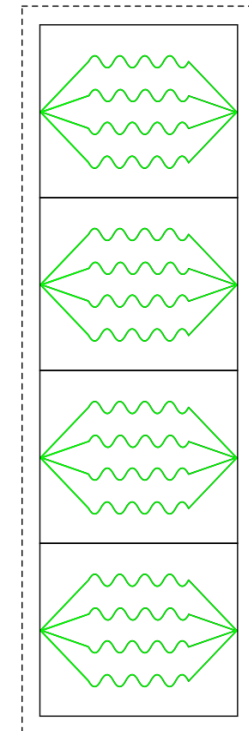
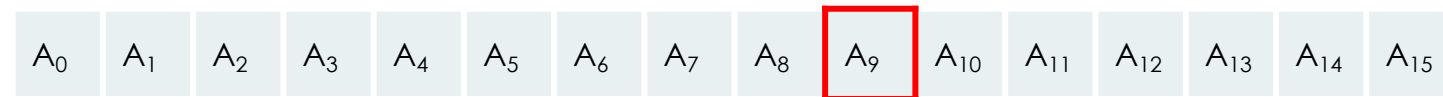
```
__global__ void vector_addition(double *a, double *b, double *c)
{
    4           2           1
    int id = blockDim.x * blockIdx.x + threadIdx.x;

    if (id < N) c[id] = a[id] + b[id];
}
```

For example, with `blockIdx.x = 2` and `threadIdx.x = 1`...



`int id = 4 * 2 + 1 = 9`



Example: Vector Addition (HIP Kernel)

GPU kernel

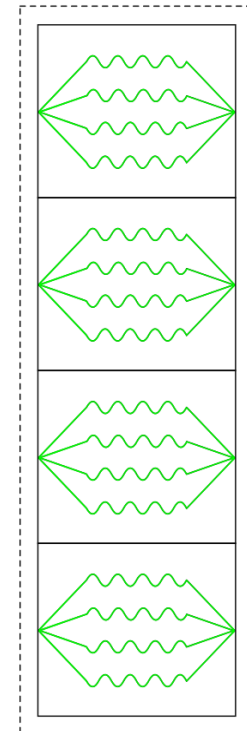
```
__global__ void vector_addition(double *a, double *b, double *c)
{
    int id = blockDim.x * blockIdx.x + threadIdx.x;

    if (id < N) c[id] = a[id] + b[id];
}
```

int id

Local variables (allocated on the stack) are private to each thread.

The loop was replaced by a grid of threads.



Example: Vector Addition (HIP Kernel)

GPU kernel

```
__global__ void vector_addition(double *a, double *b, double *c)
{
    int id = blockDim.x * blockIdx.x + threadIdx.x;

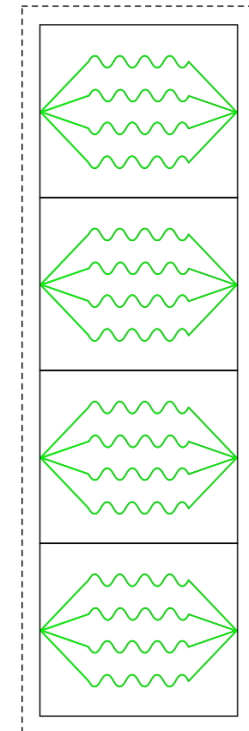
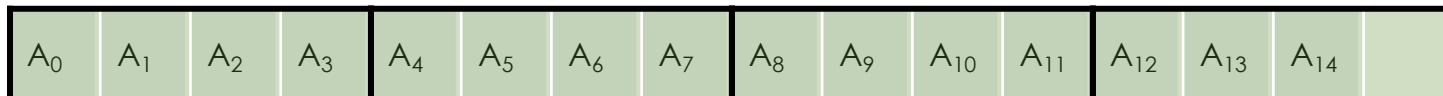
    if (id < N) c[id] = a[id] + b[id];
}
```

`if (id < N)`

```
N          = 15;
thr_per_blk = 4;
blk_in_grid = ceil(float(N) / thr_per_blk);
```

`= ceil(3.75) = 4`

Number of threads in the grid might be larger than number of elements in the array. E.g., if $N = 15$



Example: Vector Addition (HIP Kernel)

How do we call/launch the kernel?

In general

```
hipLaunchKernelGGL(<kernel_name>, <num_blocks_in_grid>, <num_threads_in_block>,  
                  <shared_memory_size>, <stream_id>,  
                  <arg0>, <arg1>, ...);
```

For our vector addition problem

```
hipLaunchKernelGGL(vector_addition, blk_in_grid, thr_per_blk,  
                  0, 0,  
                  d_a, d_b, d_c);
```

...where

```
thr_per_blk = 128;  
blk_in_grid = ceil(float(N) / thr_per_blk);
```

Clone the Repository

The repository for code examples and exercises can be found here:
https://github.com/olcf/intro_to_hip

To clone the repo (on Summit):

```
$ git clone https://github.com/olcf/intro_to_hip.git
```

NOTES:

- The `$` is not part of the command. It's meant to represent the command line prompt.
- If you have issues needing a username/password for git, it's possible you accidentally mis-typed the command above.

Example: Vector Addition (Demo 1)

Navigate to the vector addition program directory:

```
$ cd intro_to_hip/examples/vector_addition
```

Load necessary software modules and compile the code:

```
$ module load cuda/11.5.2
```

```
$ module load hip-cuda/5.1.0
```

```
$ make
```

Once you have compiled your code, submit a batch job:

```
$ bsub submit.lsf
```

You can check the status of your running job with:

```
$ jobstat -u <username>
```

Once your job has finished, check that it ran successfully by looking for the string `__SUCCESS__` in your stdout file from the job. You will also see the values of `N`, `thr_per_blk`, and `blk_in_grid`.

Example: Vector Addition (Demo 1)

What happens if you change `thr_per_blk` to be too large?

Change `thr_per_blk` to `2048` to see what happens:

```
$ vim vector_addition.cpp
```

Recompile the code: If you have already completed exercise 1, you shouldn't need to reload the modules, but you'll still need to recompile with `make`

```
$ module load cuda/11.5.2  
$ module load hip-cuda/5.1.0  
$ make
```

Once you have recompiled your code, submit a batch job:

```
$ bsub submit.lsf
```

Example: Vector Addition (Demo 1)

What happens if you change `N` to be too large?

Change `thr_per_block` back to `128`, then change `N` to `5e9` to see what happens:

```
$ vim vector_addition.cpp
```

Recompile the code: If you have already completed exercise 1, you shouldn't need to reload the modules, but you'll still need to recompile with `make`

```
$ module load cuda/11.5.2  
$ module load hip-cuda/5.1.0  
$ make
```

Once you have recompiled your code, submit a batch job:

```
$ bsub submit.lsf
```

HIP Error Checking

There are 3 main types of HIP errors

- Errors from HIP API calls
 - HIP API calls all return a `hipError_t` value that can be checked.
- Synchronous HIP kernel errors
 - These errors are related to the kernel launch
- Asynchronous HIP kernel errors
 - These errors are related to the kernel execution

HIP Error Checking

There are 2 main types of HIP errors

- Synchronous Errors
 - Errors from synchronous HIP API calls
 - HIP API calls all return a `hipError_t` value that can be checked.
 - Synchronous HIP kernel errors - these errors are related to the kernel launch
- Asynchronous Errors
 - Asynchronous HIP kernel errors
 - These errors are related to the kernel execution
 - Asynchronous HIP API calls

HIP Error Checking – API Errors

HIP API calls return a `hipError_t` value, which either reports `hipSuccess` or an error message.

```
int main()
{
    ...

    hipError_t gpuErr;

    gpuErr = hipMalloc(&d_A, bytes);

    if(hipSuccess != gpuErr){
        printf("GPU Error - %s\n", hipGetErrorString(gpuErr));
        exit(1);
    }

    ...

}
```

HIP Error Checking – API Errors

HIP API calls return a `hipError_t` value, which either reports `hipSuccess` or an error message.

- Wrap HIP API calls in error-checking macro

```
// Macro for checking errors in GPU API calls
#define gpuErrorCheck(call) \
do{ \
    hipError_t gpuErr = call; \
    if(hipSuccess != gpuErr){ \
        printf("GPU Error - %s:%d: '%s'\n", __FILE__, __LINE__, hipGetErrorString(gpuErr)); \
        exit(1); \
    } \
}while(0)

// Main program
int main()
{
    ...

    gpuErrorCheck( hipMalloc(&d_A, bytes) );
    ...

    return 0;
}
```

HIP Error Checking – Kernel Errors

HIP Kernel errors can result from kernel launch and/or kernel execution.

- Synchronous errors – e.g., kernel launch errors
- Asynchronous errors – e.g., invalid memory access (need to synchronize to find these before moving on)

```
// Launch kernel
hipLaunchKernelGGL(vector_addition, blk_in_grid, thr_per_blk , 0, 0, d_A, d_B, d_C);

// Check for synchronous errors during kernel launch (e.g. invalid execution parameters)
gpuErrorCheck( hipGetLastError() );

// Check for asynchronous errors during GPU execution (after control is returned to CPU)
gpuErrorCheck( hipDeviceSynchronize() );
```

NOTE: The `hipDeviceSynchronize` can cause performance penalty so might want to add `debug` macro.

Example: Vector Addition w/Error Checks (Demo 2)

With proper error checking in place, let's retry our tests from exercises 2 and 3...

Recall from exercise 2, we were trying to answer the question “what happens if you change `thr_per_blk` to be too large?”

First, navigate to `intro_to_hip/examples/vector_addition_with_error_check`

Change `thr_per_blk` to 2048 to see what happens:

```
$ vim vector_addition.cpp
```

Recompile the code: If you have already completed exercise 1, you shouldn't need to reload the modules, but you'll still need to recompile with `make`

```
$ module load cuda/11.5.2
$ module load hip-cuda/5.1.0
$ make
```

Once you have recompiled your code, submit a batch job:

```
$ bsub submit.lsf
```

Once your job has finished, check for an error message in your stdout file. Why did this job fail?

Example: Vector Addition w/Error Checks (Demo 2)

With proper error checking in place, let's retry our tests from exercises 2 and 3...

Recall from exercise 3, we were trying to answer the question “what happens if you change `N` to be too large?”

First, navigate to `intro_to_hip/examples/vector_addition_with_error_check`

Change `thr_per_blk` back to `128`, then change `N` to `5e9` to see what happens:

```
$ vim vector_addition.cpp
```

Recompile the code: If you have already completed exercise 1, you shouldn't need to reload the modules, but you'll still need to recompile with `make`

```
$ module load cuda/11.5.2
$ module load hip-cuda/5.1.0
$ make
```

Once you have recompiled your code, submit a batch job:

```
$ bsub submit.lsf
```

Once your job has finished, check for an error message in your stdout file. Why did this job fail?

Timing GPU Operations with HIP Events

- A **HIP stream** is a sequence of GPU operations that is carried out in order on a GPU.
- **HIP events** can be placed into a HIP stream to time GPU operations.

```
// Create start/stop event objects and variable for elapsed time in ms
hipEvent_t start, stop;
gpuErrorCheck( hipEventCreate(&start) );
gpuErrorCheck( hipEventCreate(&stop) );
float elapsed_time_ms;

...

gpuErrorCheck( hipEventRecord(start, NULL) );

// GPU Operation(s) go here.

gpuErrorCheck( hipEventRecord(stop, NULL) );

// Possible work on CPU while GPU is churning

gpuErrorCheck( hipEventSynchronize(stop) );
gpuErrorCheck( hipEventElapsedTime(&elapsed_time_ms, start, stop) );

...

gpuErrorCheck( hipEventDestroy(start) );
gpuErrorCheck( hipEventDestroy(stop) );
```

NOTE: Synchronization is needed to make sure **stop** event has taken place before using it to calculate elapsed time.

Timing GPU Operations with HIP Events (Demo 3)

Navigate to the vector addition program directory:

```
$ cd intro_to_hip/examples/vector_addition_timing
```

Load necessary software modules and compile the code:

```
$ module load cuda/11.5.2
```

```
$ module load hip-cuda/5.1.0
```

```
$ make
```

Once you have compiled your code, submit a batch job:

```
$ bsub submit.lsf
```

You can check the status of your running job with:

```
$ jobstat -u <username>
```

Once your job has finished, check that it ran successfully by looking for the string `__SUCCESS__` in your stdout file from the job. You will also see the values of `N`, `thr_per_blk`, and `blk_in_grid`.

Now you will also see the time taken for the kernel calculations.

Multidimensional GPU Grids

In previous 1D example

```
thr_per_blk = 128
```

```
blk_in_grid = ceil( float(N) / thr_per_blk );
```

```
hipLaunchKernelGGL(vector_addition, blk_in_grid, thr_per_blk , 0, 0, d_A, d_B, d_C);
```

In general

```
dim3 threads_per_block( threads per block in x-dim,  
                        threads per block in y-dim,  
                        threads per block in z-dim);
```

```
dim3 blocks_in_grid( grid blocks in x-dim,  
                    grid blocks in y-dim,  
                    grid blocks in z-dim );
```

`dim3` is built-in c struct with member variables x, y, z

Multidimensional GPU Grids

In previous 1D example

```
thr_per_blk = 128  
blk_in_grid = ceil( float(N) / thr_per_blk );  
hipLaunchKernelGGL(vector_addition, blk_in_grid, thr_per_blk , 0, 0, d_A, d_B, d_C);
```

So we could have used

```
dim3 threads_per_block( 128, 1, 1 );  
dim3 blocks_in_grid( ceil(float(N) / threads_per_block.x), 1, 1 );  
hipLaunchKernelGGL(vector_addition, blocks_in_grid, threads_per_block , 0, 0, d_A, d_B, d_C);
```

`dim3` is built-in c struct with member variables x, y, z

Multidimensional GPU Grids – 7x10 Matrix Example

A _{0,0}	A _{0,1}	A _{0,2}	A _{0,3}	A _{0,4}	A _{0,5}	A _{0,6}	A _{0,7}	A _{0,8}	A _{0,9}		
A _{1,0}	A _{1,1}	A _{1,2}	A _{1,3}	A _{1,4}	A _{1,5}	A _{1,6}	A _{1,7}	A _{1,8}	A _{1,9}		
A _{2,0}	A _{2,1}	A _{2,2}	A _{2,3}	A _{2,4}	A _{2,5}	A _{2,6}	A _{2,7}	A _{2,8}	A _{2,9}		
A _{3,0}	A _{3,1}	A _{3,2}	A _{3,3}	A _{3,4}	A _{3,5}	A _{3,6}	A _{3,7}	A _{3,8}	A _{3,9}		
A _{4,0}	A _{4,1}	A _{4,2}	A _{4,3}	A _{4,4}	A _{4,5}	A _{4,6}	A _{4,7}	A _{4,8}	A _{4,9}		
A _{5,0}	A _{5,1}	A _{5,2}	A _{5,3}	A _{5,4}	A _{5,5}	A _{5,6}	A _{5,7}	A _{5,8}	A _{5,9}		
A _{6,0}	A _{6,1}	A _{6,2}	A _{6,3}	A _{6,4}	A _{6,5}	A _{6,6}	A _{6,7}	A _{6,8}	A _{6,9}		

M = 7 rows

N = 10 columns

Assume a 4x4 block of threads...

Then to cover all elements in the array,
we need 3 blocks in x-dim and 2 blocks in
y-dim.

```
dim3 threads_per_block( 4, 4, 1 );
```

```
dim3 blocks_in_grid( ceil( float(N) / threads_per_block.x ),
```

$\text{ceil}(10/4) = \text{ceil}(2.5) = 3$

```
        ceil( float(M) / threads_per_block.y ) , 1 );
```

$\text{ceil}(7/4) = \text{ceil}(1.75) = 2$

```
hipLaunchKernelGGL(matrix_addition, blocks_in_grid, threads_per_block , 0, 0, d_A, d_B, d_C);
```

Multidimensional GPU Grids – 7x10 Matrix Example

A _{0,0}	A _{0,1}	A _{0,2}	A _{0,3}	A _{0,4}	A _{0,5}	A _{0,6}	A _{0,7}	A _{0,8}	A _{0,9}		
A _{1,0}	A _{1,1}	A _{1,2}	A _{1,3}	A _{1,4}	A _{1,5}	A _{1,6}	A _{1,7}	A _{1,8}	A _{1,9}		
A _{2,0}	A _{2,1}	A _{2,2}	A _{2,3}	A _{2,4}	A _{2,5}	A _{2,6}	A _{2,7}	A _{2,8}	A _{2,9}		
A _{3,0}	A _{3,1}	A _{3,2}	A _{3,3}	A _{3,4}	A _{3,5}	A _{3,6}	A _{3,7}	A _{3,8}	A _{3,9}		
A _{4,0}	A _{4,1}	A _{4,2}	A _{4,3}	A _{4,4}	A _{4,5}	A _{4,6}	A _{4,7}	A _{4,8}	A _{4,9}		
A _{5,0}	A _{5,1}	A _{5,2}	A _{5,3}	A _{5,4}	A _{5,5}	A _{5,6}	A _{5,7}	A _{5,8}	A _{5,9}		
A _{6,0}	A _{6,1}	A _{6,2}	A _{6,3}	A _{6,4}	A _{6,5}	A _{6,6}	A _{6,7}	A _{6,8}	A _{6,9}		

So, how do we handle this 2D grid in our kernel?

- First, let's look at how we handle the “global thread IDs”
- Then we'll look at extra considerations to guard against stepping on memory that doesn't belong to the kernel.

```
__global__ void matrix_addition(double *a, double *b, double *c){  
    int column = blockDim.x * blockIdx.x + threadIdx.x;  
    int row     = blockDim.y * blockIdx.y + threadIdx.y;  
    if (row < M && column < N){  
        int thread_id = row * N + column;  
        c[thread_id] = a[thread_id] + b[thread_id];  
    }  
}
```

	0	1	2	3	4	5	6	7	8	9	10	11
0	A _{0,0}	A _{0,1}	A _{0,2}	A _{0,3}	A _{0,4}	A _{0,5}	A _{0,6}	A _{0,7}	A _{0,8}	A _{0,9}		
1	A _{1,0}	A _{1,1}	A _{1,2}	A _{1,3}	A _{1,4}	A _{1,5}	A _{1,6}	A _{1,7}	A _{1,8}	A _{1,9}		
2	A _{2,0}	A _{2,1}	A _{2,2}	A _{2,3}	A _{2,4}	A _{2,5}	A _{2,6}	A _{2,7}	A _{2,8}	A _{2,9}		
3	A _{3,0}	A _{3,1}	A _{3,2}	A _{3,3}	A _{3,4}	A _{3,5}	A _{3,6}	A _{3,7}	A _{3,8}	A _{3,9}		
4	A _{4,0}	A _{4,1}	A _{4,2}	A _{4,3}	A _{4,4}	A _{4,5}	A _{4,6}	A _{4,7}	A _{4,8}	A _{4,9}		
5	A _{5,0}	A _{5,1}	A _{5,2}	A _{5,3}	A _{5,4}	A _{5,5}	A _{5,6}	A _{5,7}	A _{5,8}	A _{5,9}		
6	A _{6,0}	A _{6,1}	A _{6,2}	A _{6,3}	A _{6,4}	A _{6,5}	A _{6,6}	A _{6,7}	A _{6,8}	A _{6,9}		
7												

M = 7 rows

N = 10 columns

Assume 4x4 blocks of threads...

Then to cover all elements in the array, we need 3 blocks in x-dim and 2 blocks in y-dim.

```
__global__ void matrix_addition(double *a, double *b, double *c){
    int column = blockDim.x * blockIdx.x + threadIdx.x; (0 - 11)
    int row     = blockDim.y * blockIdx.y + threadIdx.y; (0 - 7)
    if (row < M && column < N){
        int thread_id = row * N + column; (0 - 69)
        c[thread_id] = a[thread_id] + b[thread_id];
    }
}
```

Ex: What element of the array does the highlighted thread correspond to?

$$\begin{aligned} \text{thread_id} &= \text{row} * N + \text{column} \\ &= 5 * 10 + 6 = 56 \end{aligned}$$

Hands-On Exercise 1: Find the Error

Using the reported error message, identify the problem and fix it.

First, navigate to `intro_to_hip/exercises/vector_addition_find_the_error`

Then compile the code: If you have already completed previous exercises, you shouldn't need to reload the modules, but you'll still need to compile with `make`

```
$ module load cuda/11.5.2
$ module load hip-cuda/5.1.0
$ make
```

Once you have compiled your code, submit a batch job:

```
$ bsub submit.lsf
```

Once your job has finished, look for an error message in the stdout file and use the error (as well as the source code in `vector_addition.cpp`) to identify the problem.

Then fix the problem (in `vector_addition.cpp`), recompile the code (`make`), and re-run it (`bsub submit.lsf`).

If the new stdout file reports `__SUCCESS__`, the problem has been solved.

Hands-On Exercise 2: Complete the Kernel

The kernel function is missing the actual calculation. Add it.

First, navigate to `intro_to_hip/square_array_elements_complete_kernel`

The file `square_array_elements.cpp` contains a program which simply takes an array and squares each of the elements on the GPU. However, the “squaring of the array elements” is missing from the GPU kernel.

Add the missing calculation in the `square_array_elements.cpp` file.

Then compile the code: If you have already completed previous exercises, you shouldn't need to reload the modules, but you'll still need to compile with `make`

```
$ module load cuda/11.5.2
$ module load hip-cuda/5.1.0
$ make
```

Once you have compiled your code, submit a batch job:

```
$ bsub submit.lsf
```

If the new stdout file reports `__SUCCESS__`, you have correctly added the missing calculations.

Hands-On Exercise 3: Time the GPU Kernel with HIP events

Add HIP events to time the duration of the GPU kernel.

First, navigate to `intro_to_hip/square_array_elements_time_kernel`

The file `square_array_elements.cpp` contains a program which takes an array and squares each of the elements on the GPU.

Add HIP events to time the kernel in the `square_array_elements.cpp` file and print out the duration.

Then compile the code: If you have already completed previous exercises, you shouldn't need to reload the modules, but you'll still need to compile with `make`

```
$ module load cuda/11.5.2
$ module load hip-cuda/5.1.0
$ make
```

Once you have compiled your code, submit a batch job:

```
$ bsub submit.lsf
```

NOTE: The stdout file will already report `__SUCCESS__`, so for this exercise you'll need to look for the time duration you print out.

Hands-On Exercise 4: Complete Kernel & Data Transfer

Once again, the kernel function is missing the actual calculation, but this time in 2D. Add in the calculation. There are also arguments missing from one of the data transfers. Add them.

First, navigate to `intro_to_hip/square_matrix_elements`

The file `square_matrix_elements.cpp` contains a program which takes a matrix (2D array) and squares each of the elements on the GPU. However, the “squaring of the matrix elements” is missing from the GPU kernel.

Add the missing calculation in the `square_array_elements.cpp` file.

You will also need to add in 2 missing arguments from one of the data transfers.

Once you’ve done both TODOs, compile the code: If you have already completed previous exercises, you shouldn’t need to reload the modules, but you’ll still need to compile with `make`

```
$ module load cuda/11.5.2
$ module load hip-cuda/5.1.0
$ make
```

Once you have compiled your code, submit a batch job:

```
$ bsub submit.lsf
```

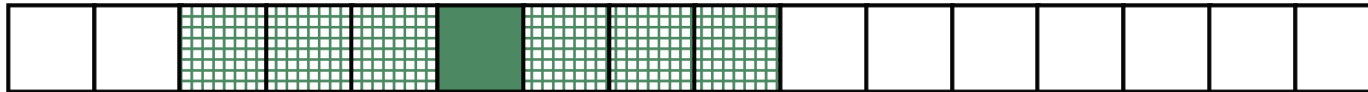
If the stdout file reports `__SUCCESS__`, you have correctly added the missing calculations and arguments.

Optimizing an Example HIP Code

Pinned Memory, Asynchronous Data Transfer, and Overlapping CPU and GPU work.

Let's take a look at an example HIP code to demonstrate some simple optimizations...

Example: For each element in an array, calculates the average value of that element, the 3 elements to its left, and the 3 elements to its right.



- The program computes the averages on the GPU and compares with a CPU implementation for correctness.
- The CPU version is already sped up with OpenMP CPU threading.

Optimizing an Example HIP Code

Pinned Memory, Asynchronous Data Transfer, and Overlapping CPU and GPU work.

- Kernel execution is asynchronous w.r.t. the CPU.
- Asynchronous data transfers

```
double *A          = (double*)malloc(bytes);
double *A_average_gpu = (double*)malloc(bytes);
...
average_on_cpu();

hipErrorCheck( hipMemcpy(d_A, A, bytes, hipMemcpyHostToDevice) );

int thr_per_blk = block_size;
int blk_in_grid = ceil( float(N+2*stencil_radius) / thr_per_blk );

hipLaunchKernelGGL(average_array_elements, blk_in_grid, thr_per_blk, 0, 0, d_A, d_A_average);

hipErrorCheck( hipMemcpy(A_average_gpu, d_A_average, bytes, hipMemcpyDeviceToHost) );
```

PLAN:

- Make all GPU operations asynchronous
- Move `average_on_cpu()` after asynchronous GPU commands
 - While GPU operations are off being performed in the HIP stream, the CPU calculation can be performed concurrently.

Optimizing an Example HIP Code

Pinned Memory, **Asynchronous Data Transfers**, and Overlapping CPU and GPU work.

Performs asynchronous GPU data transfers

```
hipError_t hipMemcpyAsync(void* dst, const void* src, size_t bytes, hipMemcpyKind kind, hipStream_t stream)
```

NOTE: Must use pinned (page-locked) CPU memory for asynchronous data transfers.



Allocates page-locked host memory

```
hipError_t hipHostMalloc(void **dst, size_t size, unsigned int flags)
```

Optimizing an Example HIP Code

Pinned Memory, Asynchronous Data Transfers, and Overlapping CPU and GPU work.

```
double *A          = (double*)malloc(bytes);
double *A_average_gpu = (double*)malloc(bytes);
...
average_on_cpu();

hipErrorCheck( hipMemcpy(d_A, A, bytes, hipMemcpyHostToDevice) );

int thr_per_blk = block_size;
int blk_in_grid = ceil( float(N+2*stencil_radius) / thr_per_blk );

hipLaunchKernelGGL(average_array_elements, blk_in_grid, thr_per_blk, 0, 0, d_A, d_A_average);

hipErrorCheck( hipMemcpy(A_average_gpu, d_A_average, bytes, hipMemcpyDeviceToHost) );
```

Pageable



Pinned

```
double *A, *A_average_gpu;
hipErrorCheck( hipHostMalloc(&A, bytes) );
hipErrorCheck( hipHostMalloc(&A_average_gpu, bytes) );
...
average_on_cpu();

hipErrorCheck( hipMemcpy(d_A, A, bytes, hipMemcpyHostToDevice) );

int thr_per_blk = block_size;
int blk_in_grid = ceil( float(N+2*stencil_radius) / thr_per_blk );

hipLaunchKernelGGL(average_array_elements, blk_in_grid, thr_per_blk, 0, 0, d_A, d_A_average);

hipErrorCheck( hipMemcpy(A_average_gpu, d_A_average, bytes, hipMemcpyDeviceToHost) );
```

Use pinned memory in place of pageable memory: ~35% speedup

Optimizing an Example HIP Code

Pinned Memory, **Asynchronous Data Transfers**, and Overlapping CPU and GPU work.

```
double *A, *A_average_gpu;
hipErrorCheck( hipHostMalloc(&A, bytes) );
hipErrorCheck( hipHostMalloc(&A_average_gpu, bytes) );
...
average_on_cpu();

hipErrorCheck( hipMemcpy(d_A, A, bytes, hipMemcpyHostToDevice) );

int thr_per_blk = block_size;
int blk_in_grid = ceil( float(N+2*stencil_radius) / thr_per_blk );

hipLaunchKernelGGL(average_array_elements, blk_in_grid, thr_per_blk, 0, 0, d_A, d_A_average);

hipErrorCheck( hipMemcpy(A_average_gpu, d_A_average, bytes, hipMemcpyDeviceToHost) );
```

Synchronous



Asynchronous

```
double *A, *A_average_gpu;
hipErrorCheck( hipHostMalloc(&A, bytes) );
hipErrorCheck( hipHostMalloc(&A_average_gpu, bytes) );
...
average_on_cpu();

hipErrorCheck( hipMemcpyAsync(d_A, A, bytes, hipMemcpyHostToDevice, 0) );

int thr_per_blk = block_size;
int blk_in_grid = ceil( float(N+2*stencil_radius) / thr_per_blk );

hipLaunchKernelGGL(average_array_elements, blk_in_grid, thr_per_blk, 0, 0, d_A, d_A_average);

hipErrorCheck( hipMemcpyAsync(A_average_gpu, d_A_average, bytes, hipMemcpyDeviceToHost, 0) );

hipErrorCheck( hipDeviceSynchronize() );
```

Optimizing an Example HIP Code

Pinned Memory, Asynchronous Data Transfers, and **Overlapping CPU and GPU work.**

```
double *A, *A_average_gpu;
hipErrorCheck( hipHostMalloc(&A, bytes) );
hipErrorCheck( hipHostMalloc(&A_average_gpu, bytes) );
...

hipErrorCheck( hipMemcpyAsync(d_A, A, bytes, hipMemcpyHostToDevice, 0) );


int thr_per_blk = block_size;
int blk_in_grid = ceil( float(N+2*stencil_radius) / thr_per_blk );

hipLaunchKernelGGL(average_array_elements, blk_in_grid, thr_per_blk, 0, 0, d_A, d_A_average);

hipErrorCheck( hipMemcpyAsync(A_average_gpu, d_A_average, bytes, hipMemcpyDeviceToHost, 0) );

average_on_cpu();

hipErrorCheck( hipDeviceSynchronize() );
```



Moved CPU calculation after the asynchronous kernel launch and data transfer.

Optimizing an Example HIP Code

Pinned Memory, Asynchronous Data Transfer, and Overlapping CPU and GPU work.

Code Version	CPU Time (s)	GPU Time (s)	Total Time (s)	Total Time Speedup over previous	Total Time Speedup overall
Pageable	0.1916	0.0015	0.2863	1	1
Pinned	0.1948	0.0015	0.2192	1.31x (~31%)	1.31x (~31%)
Pinned+Overlap	0.1934	0.0014	0.2064	1.06x (~6%)	1.39x (~39%)

- Using page-locked host memory gave the biggest boost of 31% (because data transfers were 4X faster)
- Then, by adding asynchronous data transfers and running the GPU operations at the same time as the CPU calculations, the time for the GPU operations was “hidden” behind the CPU time (total time → CPU time only)

Exercise 5: Matrix Multiply (1D indexing)

	0	1	2	3	4	5	6		0	1	2	3	4	5	6		0	1	2	3	4	5	6
0	A ₀	A ₁	A ₂	A ₃	A ₄	A ₅	A ₆		B ₀	B ₁	B ₂	B ₃	B ₄	B ₅	B ₆		C ₀	C ₁	C ₂	C ₃	C ₄	C ₅	C ₆
1	A ₇	A ₈	A ₉	A ₁₀	A ₁₁	A ₁₂	A ₁₃		B ₇	B ₈	B ₉	B ₁₀	B ₁₁	B ₁₂	B ₁₃		C ₇	C ₈	C ₉	C ₁₀	C ₁₁	C ₁₂	C ₁₃
2	A ₁₄	A ₁₅	A ₁₆	A ₁₇	A ₁₈	A ₁₉	A ₂₀		B ₁₄	B ₁₅	B ₁₆	B ₁₇	B ₁₈	B ₁₉	B ₂₀		C ₁₄	C ₁₅	C ₁₆	C ₁₇	C ₁₈	C ₁₉	C ₂₀
3	A ₂₁	A ₂₂	A ₂₃	A ₂₄	A ₂₅	A ₂₆	A ₂₇		B ₂₁	B ₂₂	B ₂₃	B ₂₄	B ₂₅	B ₂₆	B ₂₇	=	C ₂₁	C ₂₂	C ₂₃	C ₂₄	C ₂₅	C ₂₆	C ₂₇
4	A ₂₈	A ₂₉	A ₃₀	A ₃₁	A ₃₂	A ₃₃	A ₃₄		B ₂₈	B ₂₉	B ₃₀	B ₃₁	B ₃₂	B ₃₃	B ₃₄		C ₂₈	C ₂₉	C ₃₀	C ₃₁	C ₃₂	C ₃₃	C ₃₄
5	A ₃₅	A ₃₆	A ₃₇	A ₃₈	A ₃₉	A ₄₀	A ₄₁		B ₃₅	B ₃₆	B ₃₇	B ₃₈	B ₃₉	B ₄₀	B ₄₁		C ₃₅	C ₃₆	C ₃₇	C ₃₈	C ₃₉	C ₄₀	C ₄₁
6	A ₄₂	A ₄₃	A ₄₄	A ₄₅	A ₄₆	A ₄₇	A ₄₈		B ₄₂	B ₄₃	B ₄₄	B ₄₅	B ₄₆	B ₄₇	B ₄₈		C ₄₂	C ₄₃	C ₄₄	C ₄₅	C ₄₆	C ₄₇	C ₄₈

$$C_8 = A_7 * B_1 + A_8 * B_8 + A_9 * B_{15} + A_{10} * B_{22} + A_{11} * B_{29} + A_{12} * B_{36} + A_{13} * B_{43}$$

```
A_index = row * N + i
B_index = i * N + col
C_index = row * N + col
```

```
A_index = row * 7 + i    = 1 * 7 + i = 7 + i
B_index = i * 7 + col = i * 7 + 1 = i * 7 + 1
C_index = 1 * 7 + 1    = 8                = 8
```

(where N = 7 and i = {0..(N-1)})

Exercise 5: Matrix Multiply (simple implementation)

	0	1	2	3	4	5	6	7
0	A ₀	A ₁	A ₂	A ₃	A ₄	A ₅	A ₆	
1	A ₇	A ₈	A ₉	A ₁₀	A ₁₁	A ₁₂	A ₁₃	
2	A ₁₄	A ₁₅	A ₁₆	A ₁₇	A ₁₈	A ₁₉	A ₂₀	
3	A ₂₁	A ₂₂	A ₂₃	A ₂₄	A ₂₅	A ₂₆	A ₂₇	
4	A ₂₈	A ₂₉	A ₃₀	A ₃₁	A ₃₂	A ₃₃	A ₃₄	
5	A ₃₅	A ₃₆	A ₃₇	A ₃₈	A ₃₉	A ₄₀	A ₄₁	
6	A ₄₂	A ₄₃	A ₄₄	A ₄₅	A ₄₆	A ₄₇	A ₄₈	
7								

Your task for this exercise is to write a simple implementation of the matrix multiply kernel, where each thread calculates one element of the output matrix.

```
__global__ void matrix_multiply(double *a, double *b, double *c)
{
    int column = blockDim.x * blockIdx.x + threadIdx.x;
    int row    = blockDim.y * blockIdx.y + threadIdx.y;

    // TODO: Add matrix multiply implementation here
}
```

NOTE that `row` and `column` refer to the row and column of the GPU grid here (this is slightly different than the previous slide), which can be larger than the matrix itself.

Where to go from here?

We did not cover:

- Shared memory – will be covered in a future session on optimization techniques
- HIP streams - will be covered in a future session on optimization techniques
- Profiling – can be done w/tools from the underlying platform (Nsight on NVIDIA rocprof on AMD)
- Debugging - can be done w/tools from the underlying platform (NVIDIA or AMD)
- Managed Memory
- hipify – translating CUDA code to HIP (will be covered here <https://www.olcf.ornl.gov/calendar/hip-for-cuda-programmers/>)
- Fortran interfaces – will be covered in a future session
- HIP libraries

Resources:

- AMD's HIP Programming Guide: https://docs.amd.com/bundle/HIP-Programming-Guide-v5.2/page/Programming_with_HIP.html
- Future sessions in Preparing for Frontier training series: <https://www.olcf.ornl.gov/preparing-for-frontier/>
- HIP Github: <https://github.com/ROCm-Developer-Tools/HIP>