EE4C07: Advanced Computing Systems
GPU Hands On Slides

Imran Ashraf
Nauman Ahmed

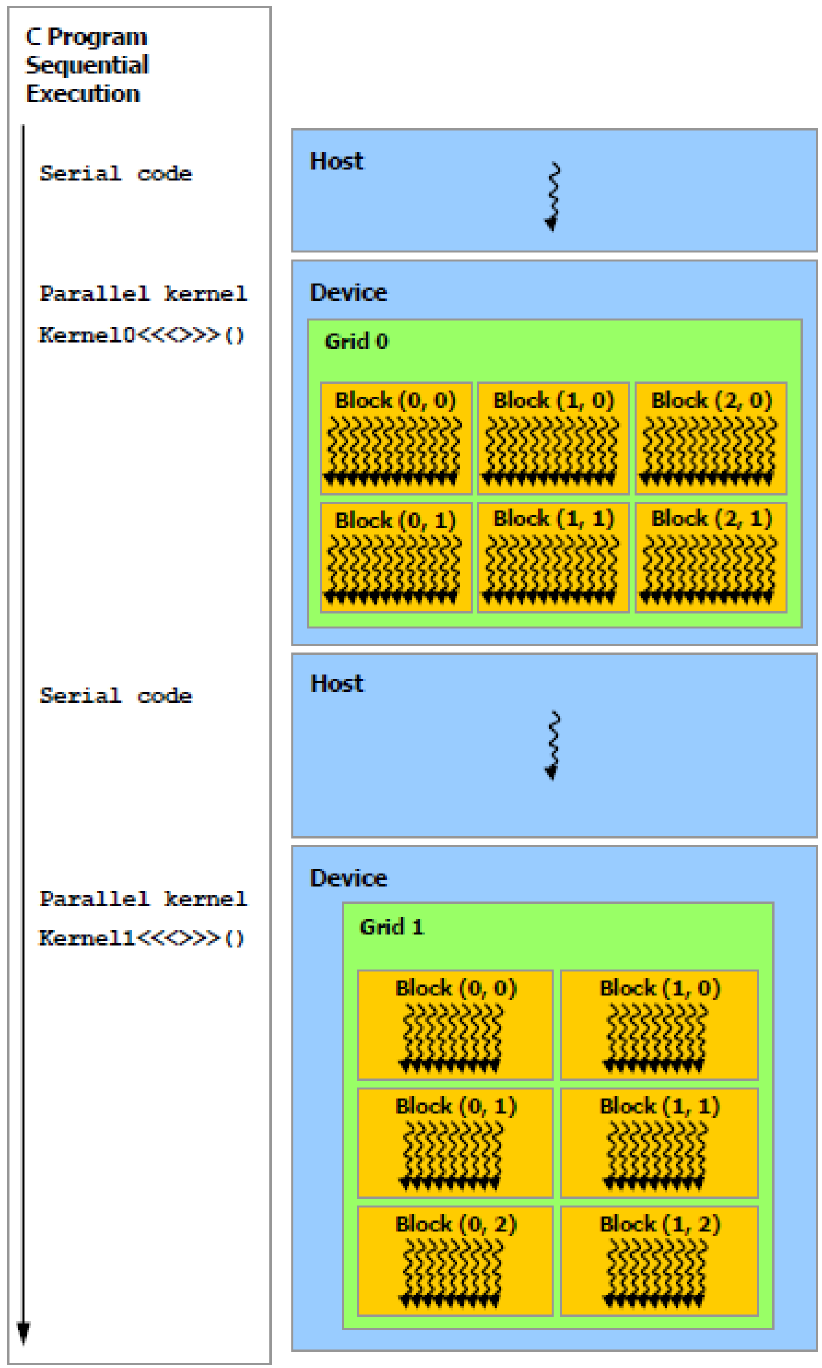
Outline

- **Setup**
- **Documentation**
- **Compilation**
- **Example Codes**
  - **CUDA**
  - **OpenCL**
  - **CUDA with Unified Memory**
  - **OpenACC**
- **Profiling/Debugging**

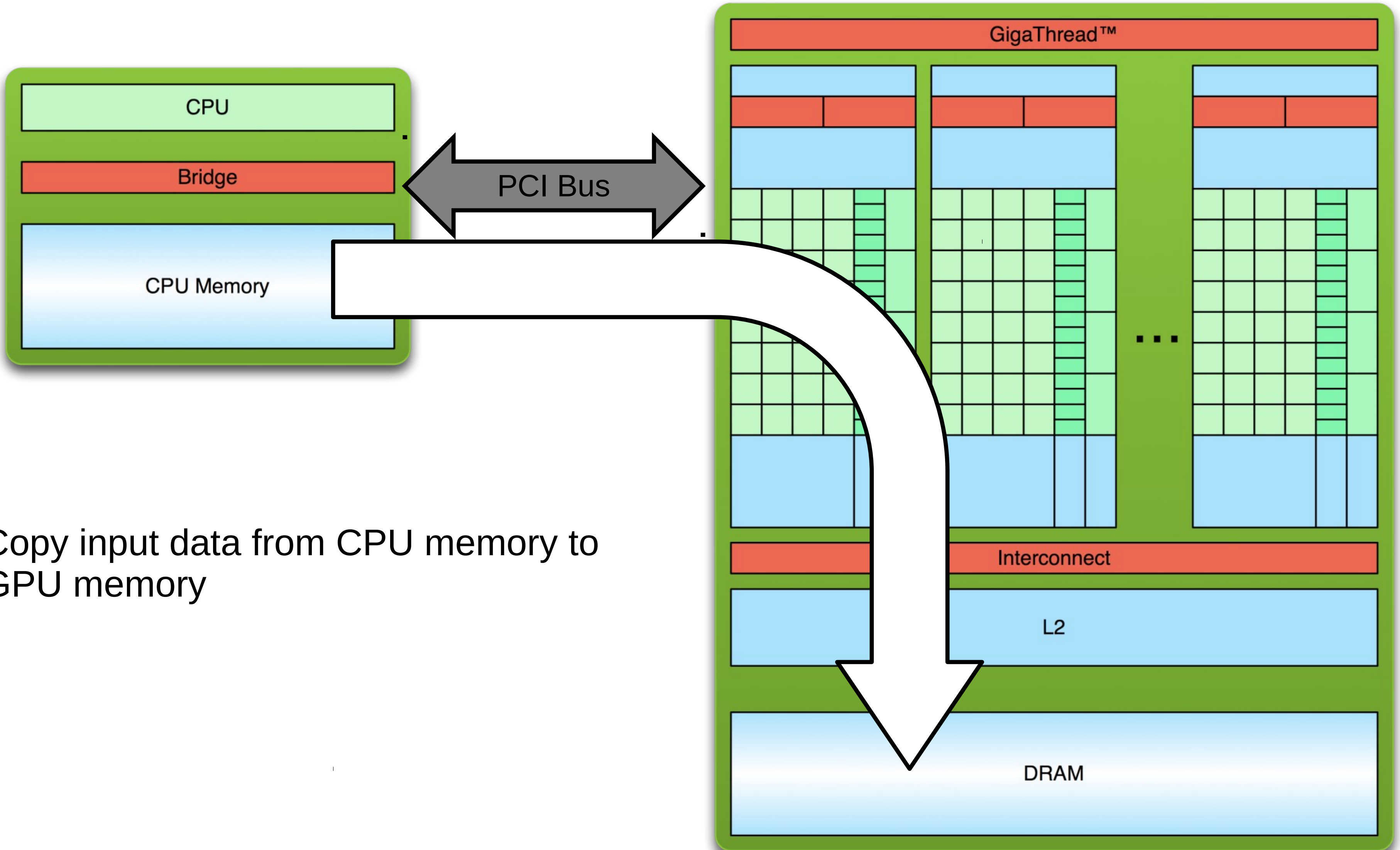
Setup

- **INSY-cluster**
- **GPU:** GeForce GTX 1080 Ti

Simple Processing Flow

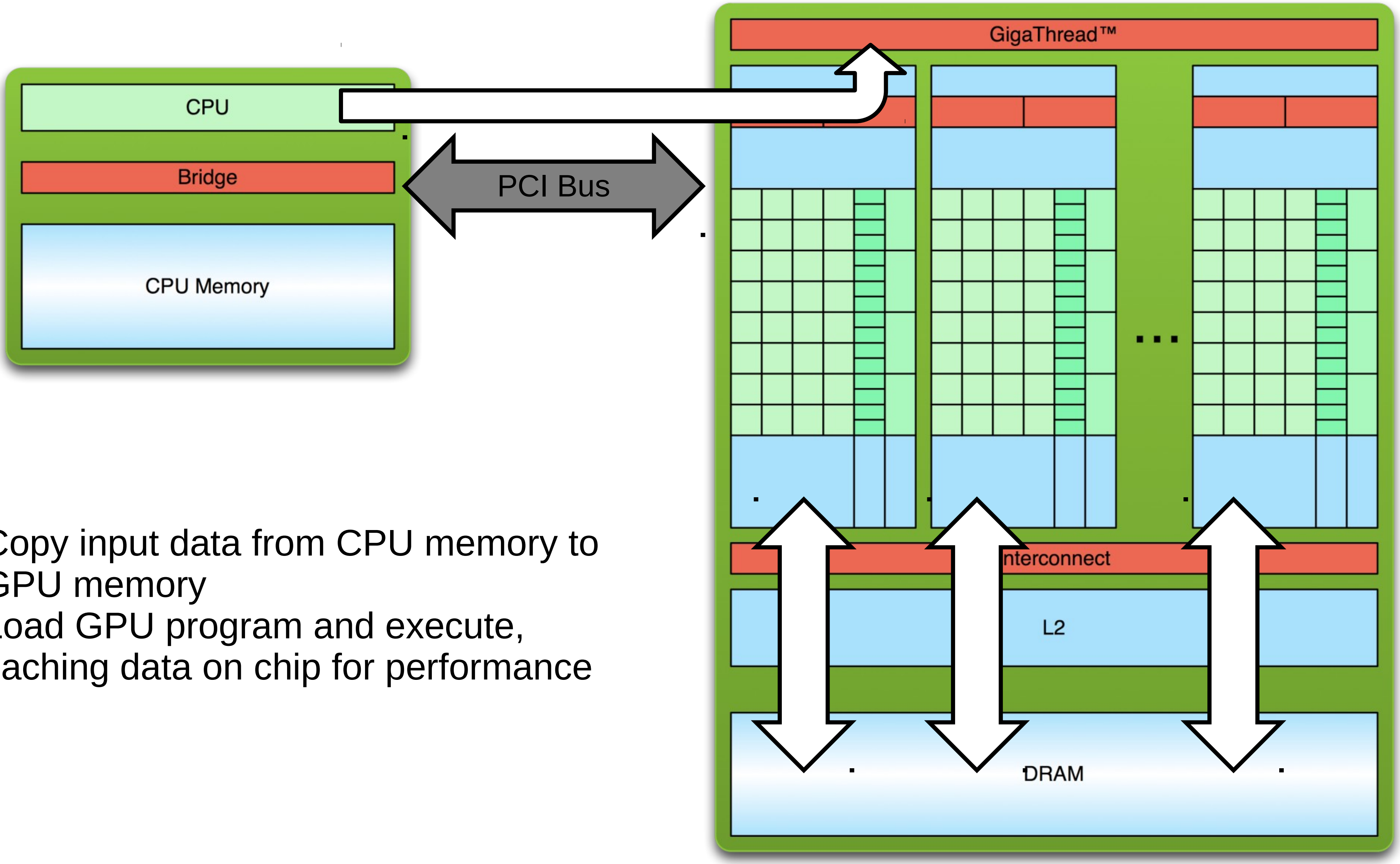


Simple Processing Flow



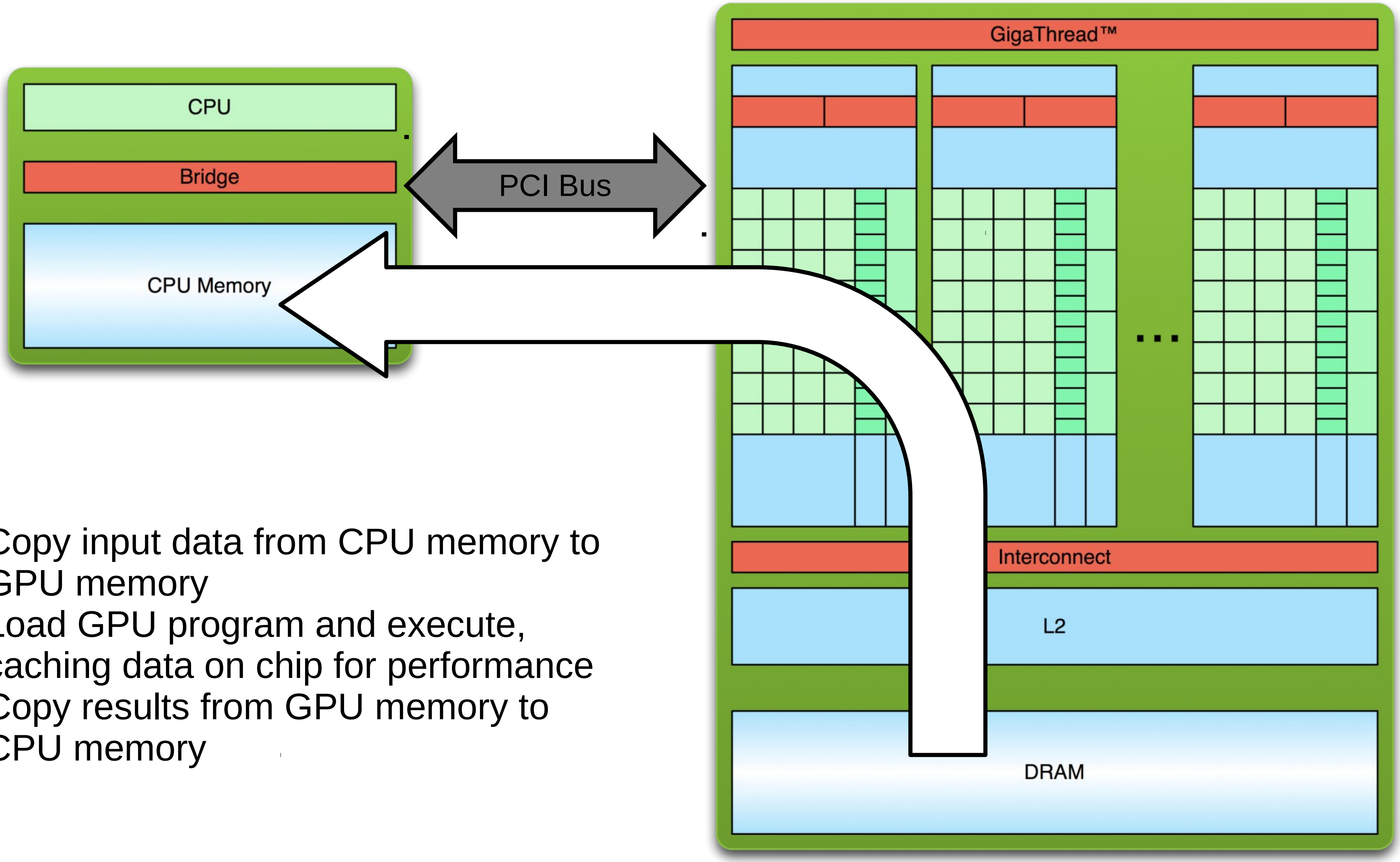
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

CUDA

- CUDA C/C++
  - Expose GPU parallelism for general-purpose computing
  - Nvidia only !!!
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - APIs to manage devices, memory etc.

Vector Add Example (CPU)

```
typedef float TYPE;

const long N =  (8*1024*1024);

TYPE *Arr1  = malloc( N*sizeof(TYPE) );

TYPE *Arr2  = malloc( N*sizeof(TYPE) );

TYPE *Sum = malloc( N*sizeof(TYPE) );

for (i = 0; i < N; i ++)  {

    Arr1[i] = 1;

    Arr2[i] = 2;

}

for (i=0; i<N; i++) {

    Sum[i] = Arr1[i] + Arr2[i];

}
```

Vector Add Example (CUDA)

int SIZE = n * sizeof(double)

- **Allocate Memory**

double * hostA = malloc (SIZE);

cudaMalloc((void **) &deviceA, SIZE);

- **Copy Data from host to device (GPU)**

cudaMemcpy(deviceA, hostA, SIZE, cudaMemcpyHostToDevice);

- **Run Kernel**

vectorAddKernel<<< GridDim, BlockDim >>>(deviceA, deviceB, deviceResult);

- **Copy Data from device to host**

cudaMemcpy(result, deviceResult, SIZE, cudaMemcpyDeviceToHost);

- **Free Resources**

cudaFree(deviceA);

Vector Add Kernel (CUDA)

```
__global__ void vectorAddKernel
    (double* deviceA, double* deviceB, double* deviceResult)
{
   unsigned index = blockIdx.x * blockDim.x + threadIdx.x;
   deviceResult[index] = deviceA[index] + deviceB[index]);
}
```

NVPROF

- To get simple timing results of the kernel launches:

  `nvprof --print-gpu-trace ./exec`

where **`exec`** is the name of the binary

- To do more elaborate profiling:

  `nvprof --export-profile vectoradd.nvprof --analysis-metrics ./exec`

this will create a file **`vectoradd.nvprof`**

- Install the same version of CUDA toolkit on your PC that you will use on the INSY cluster (latest is 10.1)

- Launch the Nvidia Visual Profiler (nvvp) on your PC. Linux command: **`nvvp`**

- Import **`vectoradd.nvprof`** as **File -> Import**

OpenCL

- OpenCL - Open Computing Language
  - Open, royalty-free standard
  - Initially proposed by Apple
  - Specification maintained by the Khronos Group
  - Developed by a number of companies
  - Specification: set of requirements to be satisfied $\Rightarrow$ must be implemented to use it
  - Device agnostic
- Framework for parallel programming across heterogeneous platforms consisting of:
  - CPUs, GPUs and other processors (FPGA, ...)
- Similar: Nvidia’s CUDA

Vector Add Host Code(OpenCL)

- **Query the system for available devices**

clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL);

- **Select which device to use**

cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);

- **Create the vectors**

float * pA = new float[4096]; float * pB = new float[4096];float * pC = new float[4096];

randomize(pA); randomize(pB);

- **Compile the kernel**

char * csProgramSource = oclLoadProgSource(VectorAdd.cl, "", KernelLength);

hProgram = clCreateProgramWithSource(hContext, 1,(const char **)&csProgramSource, &szKernelLength, &ciErr1);

- **Which kernel function to use as main()**

hKernel = clCreateKernel(hProgram, "VectorAdd", &ciErr1);

Vector Add Host Code(OpenCL)

- **Allocate GPU memory for the vectors**

```
hDeviceMemA = clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, 4048 *
    sizeof(cl_float), pA, 0);

hDeviceMemB = clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, 4048 *
    sizeof(cl_float), pB, 0);

hDeviceMemC = clCrateBuffer(hContext, CL_MEM_WRITE_ONLY , 4048 * sizeof(cl_float), 0 , 0);
```

- **Specify the kernel parameters**

```
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hDeviceMemA);

clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&hDeviceMemB);

clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&hDeviceMemC);
```

- **Run 4096 kernels (1 for each vector element)**

```
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0, 4096, 0, 0, 0, 0);
```

- **Copy results from GPU back to host memory**

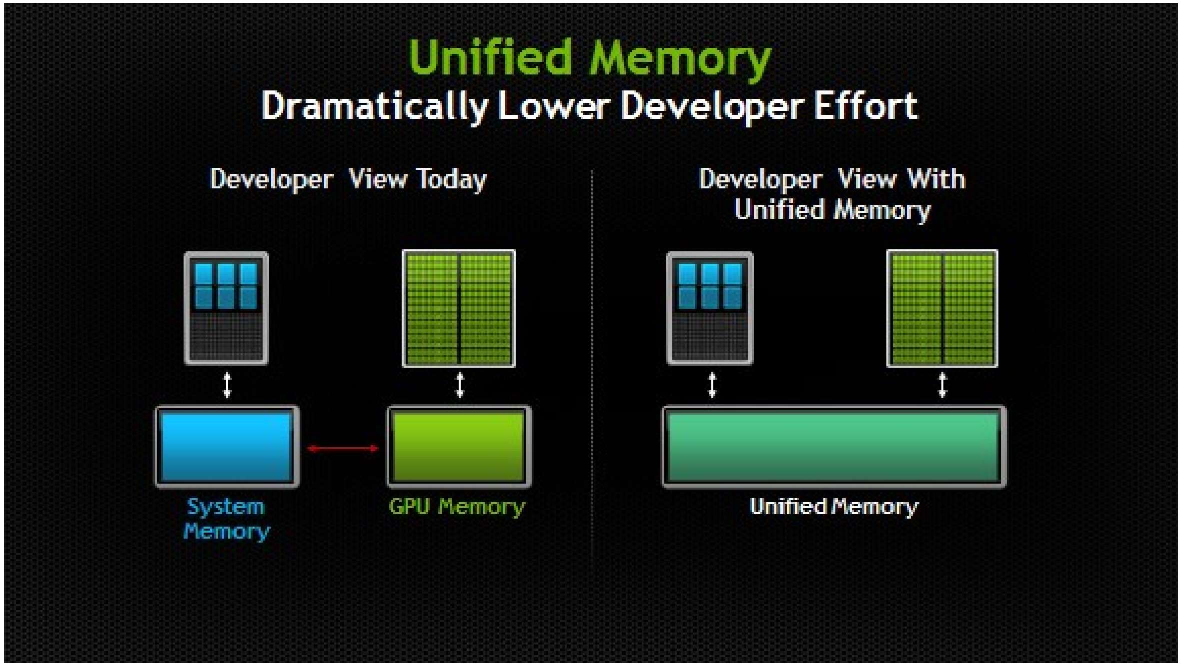
```
clEnqueueReadBuffer(hCmdQueue, hDeviceMemC, CL_TRUE, 0,4096 * sizeof(cl_float), pC, 0, 0, 0);
```

Vector Add Device Code(OpenCL)

```
__kernel void VectorAdd(__global const float* a, __global const float* b, __global float* c)
{
    // get index into global data array
    int iGID = get_global_id(0);
    // add the vector elements
    c[iGID] = a[iGID] + b[iGID];
}
```

Unified Memory (CUDA)

- Considerable programming model improvement feature
- Share pool is created and automatically managed
- Single pointer both on CPU and GPU
- Automatic data migration



Vector Add Example
(CUDA with Unified Memory)

- **Allocate Memory**

```
cudaMallocManaged(&A, SIZE);
cudaMallocManaged(&B, SIZE);
cudaMallocManaged(&C, SIZE);
```

- **Run Kernel**

```
vectorAddKernel<<< GridDim, BlockDim >>>(A, B, C);
```

- **Free Resources**

```
cudaFree(deviceA);
cudaFree(deviceA);
cudaFree(deviceA);
```

OpenACC

- OpenACC was developed by PGI, Cray, CAPS and Nvidia
- Compiler directives specify parallel regions
- OpenACC compilers handle data between host and accelerators
- Intent is to be Portable (Ind of OS, CPU/accelerators vendor)
- High-level programming: accelerator and data transfer abstraction

Vector Add Example (OpenACC)

```
TYPE *Arr1  = malloc( N*sizeof(TYPE) );
TYPE *Arr2  = malloc( N*sizeof(TYPE) );
TYPE *Sum = malloc( N*sizeof(TYPE) );

#pragma acc kernels loop copy(Arr1[0:N], Arr2[0:N])
for (i = 0; i < N; i ++)  {
    Arr1[i] = 1;
    Arr2[i] = 2;
}

#pragma acc kernels loop copy(Arr1[0:N], Arr2[0:N], Sum[0:N])
for (i=0; i<N; i++) {
    Sum[i] = Arr1[i] + Arr2[i];
}
```

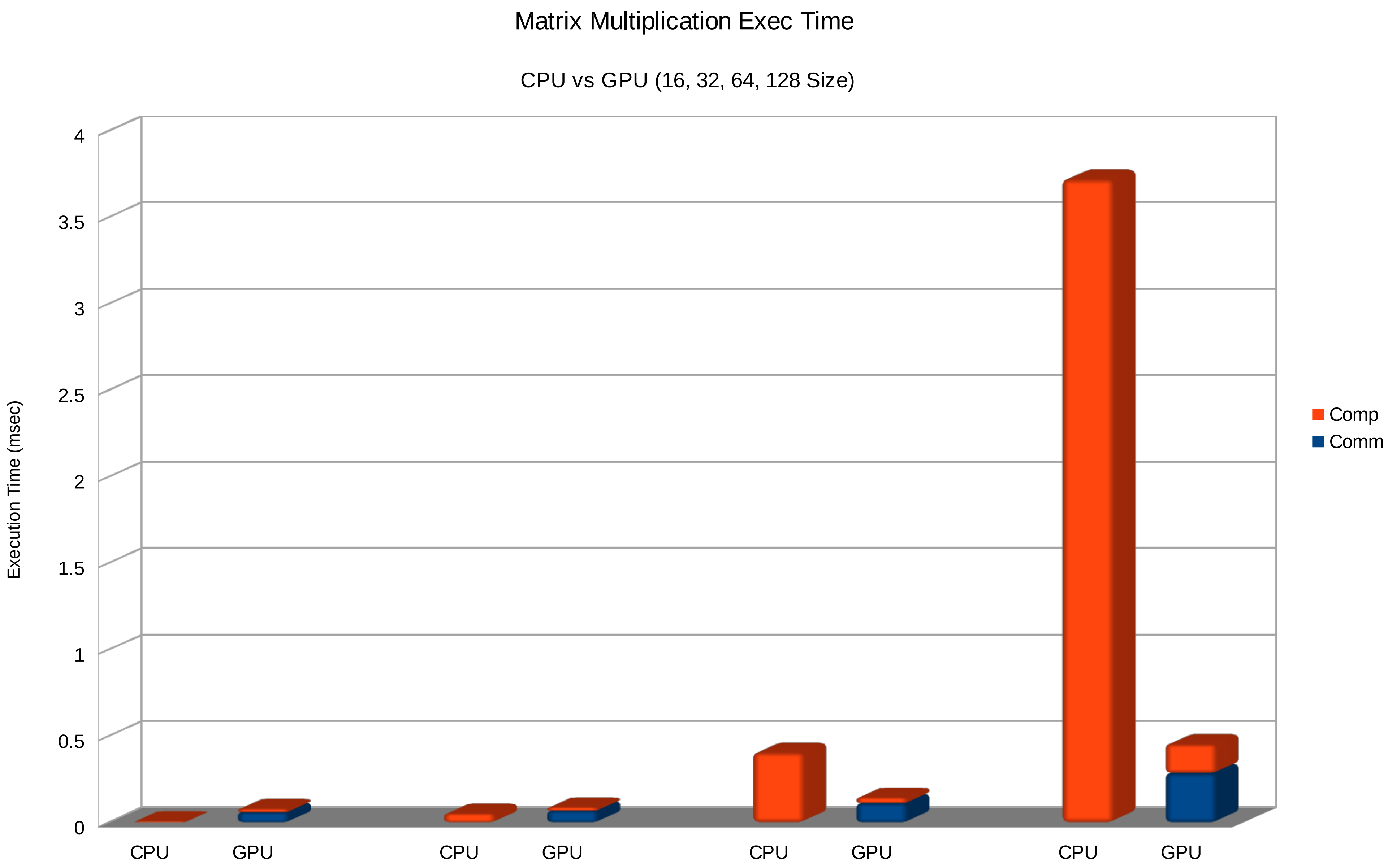
Vector Add Example
(OpenACC Optimized)

```
TYPE *Arr1  = malloc( N*sizeof(TYPE) );
TYPE *Arr2  = malloc( N*sizeof(TYPE) );
TYPE *Sum = malloc( N*sizeof(TYPE) );
#pragma acc data create(Arr1[0:N], Arr2[0:N]) copyout(Sum[0:N])
{
   #pragma acc kernels loop
    for (i = 0; i < N; i ++)  {
      Arr1[i] = 1;
      Arr2[i] = 2;
    }
   #pragma acc kernels loop
    for (i=0; i<N; i++)  {
      Sum[i] = Arr1[i] + Arr2[i];
    }
}
```

Vector Add Example
(Timings)

• CPU	:	93	msec
• GPU (CUDA)	:	51	msec (4 + 47)
• GPU (OpenACC)	:	4212	msec
• GPU (OpenACC Opt)	:	35.7	msec

Matrix Multiplication Results



Profiling

- **CUDA Occupancy Calculator**

  $CUDA_INSTALL_PATH/tools/CUDA_Occupancy_Calculator.xls

- **NVPROF**

  nvprof ./exec

- **NVVP**

  nvvp

Debugging

- **Debuggers**

  cuda-gdb

  cuda-memcheck

Questions