Introduction to Compute Unified Device Architecture (CUDA)

Kyriakos Hadjiyiannakou
What is CUDA

- **CUDA Platform**
  - Parallel computing platform and programming model invented by NVIDIA
  - Exposes the Graphics Processing Unit (GPU) parallelism for general-purpose programming
  - It enables dramatic increases in computing performance by harnessing the computing power of the GPU

- **CUDA C/C++**
  - It is based on C/C++
  - Provides extensions to enable heterogeneous programming
  - APIs to manage the execution on the GPU, moving data to and from the GPU memory, etc.
What is CUDA

- **CUDA Platform**
  - Parallel computing platform and programming model invented by NVIDIA
  - Exposes the Graphics Processing Unit (GPU) parallelism for general-purpose programming
  - It enables dramatic increases in computing performance by harnessing the computing power of the GPU

- **CUDA C/C++**
  - It is based on C/C++
  - Provides extensions to enable heterogenous programming
  - APIs to manage the execution on the GPU, moving data to and from the GPU memory, etc.
What will you learn in this session

- Write small CUDA kernels
- Compile a CUDA code
- Move data from CPU memory to GPU memory and vice versa
- Manage communication and synchronization of the execution
What will you learn in this session

- Write small CUDA kernels
- Compile a CUDA code
- Move data from CPU memory to GPU memory and vice versa
- Manage communication and synchronization of the execution

Prerequisites

- You need to be familiar with C/C++
- You do not need any experience with graphics cards
- You may need some understanding of parallel programming
- You definitely do not need any experience with pixels and graphics
- If you want to compile a CUDA code you will need to have installed the CUDA Toolkit, and an NVIDIA GPU to run it
CUDA concepts

Heterogeneous Computing

- Thread Blocks
- CUDA Threads
- Indexing of Threads
- Synchronization of Threads
- Shared Memory
- Device Memory
- Asynchronous operation
- Error Handling
- Choose Device
Heterogeneous Computing

Usual Terminology

- Host: The CPU
- Host Memory: The CPU memory
Heterogeneous Computing

**Usual Terminology**

- **Host**: The CPU
- **Host Memory**: The CPU memory
- **Device**: The GPU
- **Device Memory**: The GPU memory
Heterogeneous Computing

This is how a hybrid CPU-GPU code looks like
This is how a hybrid CPU-GPU code looks like

• We start with a serial code
• We initialize the problem we want to solve
• We transfer the data from the host to device memory
Heterogeneous Computing

This is how a hybrid CPU-GPU code looks like

- We start with a serial code
- We initialize the problem we want to solve
- We transfer the data from the host to device memory
- The host calls the GPU code
- The computationally intensive calculation is performed on the device
Heterogeneous Computing

This is how a hybrid CPU-GPU code looks like:

- We start with a serial code
- We initialize the problem we want to solve
- We transfer the data from the host to device memory

- The host calls the GPU code
- The computationally intensive calculation is performed on the device

- The host can perform other small calculations as the device is working
- After the computation the host copies the results back on the host memory
CPU vs GPU Architecture

CPU
- Low compute density
- Complex control unit
- Large caches
  - Fewer execution units (ALUs)
  - Higher clock speeds
- Small registry

GPU
- High compute Density
- High Throughput
- Can perform many computations per memory access
- Large registry
- Simple control unit
RoadMap to Pascal Arch.
## GPU Hardware

<table>
<thead>
<tr>
<th></th>
<th>Tesla K40 (Kepler)</th>
<th>Tesla P100 (Pascal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming Multiprocessors</td>
<td>15</td>
<td>56</td>
</tr>
<tr>
<td>FP32 CUDA Cores / SM</td>
<td>192</td>
<td>64</td>
</tr>
<tr>
<td>FP32 CUDA Cores / GPU</td>
<td>2880</td>
<td>3584</td>
</tr>
<tr>
<td>FP64 CUDA Cores / SM</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>FP64 CUDA Cores / GPU</td>
<td>960</td>
<td>1792</td>
</tr>
<tr>
<td>Base Clock</td>
<td>745 MHz</td>
<td>1328 MHz</td>
</tr>
<tr>
<td>Peak Perf. Single Prec.</td>
<td>4.29 Tflops</td>
<td>10.6 Tflops</td>
</tr>
<tr>
<td>Peak Perf. Double Prec.</td>
<td>1.43 Tflops</td>
<td>5.3 Tflops</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>288 GB/s</td>
<td>732 GB/s</td>
</tr>
<tr>
<td>Device Memory</td>
<td>12 GB</td>
<td>16 GB</td>
</tr>
</tbody>
</table>
Pascal GP100 SM Unit

- Warp Scheduler
- Registers
- Texture Units
- Shared Memory
Piz Daint: Europe’s fastest supercomputer

- Equipped with 4,500 nodes
- Each node has one Intel® Xeon® E5-2690 v3 @ 2.60GHz (12 cores, 64GB RAM)
- And one NVIDIA® Tesla® P100 16GB
What are CUDA blocks and threads?

- A thread block is a programming abstraction.
- For better process and data mapping, threads are grouped into blocks.
- All the blocks compose the grid of blocks.
- The CUDA blocks are distributed across the SMs.
- Threads within a block can communicate via the shared memory.
- Threads in different blocks cannot communicate.
- Each block runs on 1 SM.
<table>
<thead>
<tr>
<th></th>
<th>Kepler GK110</th>
<th>Pascal GP100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>3.5</td>
<td>6.0</td>
</tr>
<tr>
<td>Threads per Warp</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max Warps per SM</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Max Threads per SM</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Max Thread Blocks per SM</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Max Registers per block</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers per thread</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>Max Registers per SM</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Thread block size</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Shared Memory per SM</td>
<td>48 KB</td>
<td>64 KB</td>
</tr>
</tbody>
</table>
Introduction to CUDA programming

• Largely like standard C/C++ with extensions
• The **nvcc** compiler splits the code in CPU and GPU parts
  ➡ A standard compiler deals with the CPU parts
  ➡ The nvcc compiler deals only with the GPU code
• New kind of functions attributes
  ➡ **__global__** functions called by host, run on device
  ➡ **__device__** functions called by device, run on device
  ➡ **__host__** functions called on host, run on host (redundant)
  ➡ A function can be simultaneously **__host__** **__device__** which means it will be compiled for both CPU and GPU architectures
• **__global__** functions are CUDA kernels and need special arguments to call them e.g. <<<>>>
Hello World! example

```c
int main(void){
    printf(“Hello World!\n”);
    return 0;
}
```

- Standard C code that runs on the host
- `nvcc` can be used to compile it also
Hello World! example

```c
#include <stdio.h>
__global__ void helloWorld(void) {
    printf("Hello World\n");
}

int main() {
    helloWorld<<<1,1>>>() ;

    return 0;
}
```

- Standard C code that runs on the host
- `nvcc` can be used to compile it also

- Two new syntactic elements `<<<>>>`
  - Number of blocks
  - Number of threads per block
- In older architectures it was not possible to print from inside a CUDA kernel
Hello World! example

```c
int main(void){
    printf("Hello World!\n");
    return 0;
}
```

- Standard C code that runs on the host
- `nvcc` can be used to compile it also

```c
#include <stdio.h>
__global__ void helloWorld(void){
    printf("Hello World\n");
}
```

- Two new syntactic elements `<<<>>>`
  - Number of blocks
  - Number of threads per block
- In older architectures it was not possible to print from inside a CUDA kernel

```c
int main(){
    helloWorld<<<1,1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Kernel calls are asynchronous
#include <stdio.h>
__global__ void helloWorld(void)
{
    printf("Hello World from (block=%d, thread=%d)\n", blockIdx.x, threadIdx.x);
}

int main()
{
    helloWorld<<< 3, 2 >>>();
    cudaDeviceSynchronize();
    return 0;
}

• Runtime variables (take values at runtime)
  - threadIdx
  - blockIdx
  - blockDim
  - gridDim
• Blocks are executed in parallel
• The execution of the blocks is asynchronous
• Threads inside a warp are running concurrently
Vector Addition using CUDA threads

```c
#define N 512
int main(void){
    int *a, *b, *c; // host pointers
    int *d_a, *d_b, *d_c; // device pointers
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = (int *)malloc(size); random_ints(a, N); // Alloc space host, random initialization
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```
Vector Addition using CUDA threads

// Copy data from host to device memory
// cudaMemcpyHostToDevice is a flag determining copying from host to dev.
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch kernel to add two vector with N threads and 1 block
// Kernel calls are asynchronous
addVecs<<<1,N>>>(d_c, d_a, d_b);

// Copy results from device to host
// cudaMemcpy blocks CPU until Kernels finish execution
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// needs cudaFree to deallocate device pointers
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c); free(a); free(b); free(c);
return 0;
Vector Addition

1. Using 1 block and N threads

```c
__global__ void addVecs(int *c, int *a, int *b)
{
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

2. Using N blocks and 1 thread per block

```c
__global__ void addVecs(int *c, int *a, int *b)
{
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```
Vector Addition

1. Using 1 block and N threads
   
   ```c
   __global__ void addVecs(int *c, int *a, int *b){
   c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
   }
   ```

2. Using N blocks and 1 thread per block
   
   ```c
   __global__ void addVecs(int *c, int *a, int *b){
   c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
   }
   ```

3. Using n blocks and m threads per block (n*m=N)
Example: \( N=32 \)

Indexing with blocks and threads

- \( \text{threadIdx.x} \)
- \( \text{blockIdx.x} \)

For different values of \( \text{blockIdx.x} \):
- \( \text{blockIdx.x} = 0 \)
- \( \text{blockIdx.x} = 1 \)
- \( \text{blockIdx.x} = 2 \)
- \( \text{blockIdx.x} = 3 \)
Vector Addition

Example: $N=32$

Indexing with blocks and threads

```
threadIdx.x
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
blockIdx.x = 0  blockIdx.x = 1  blockIdx.x = 2  blockIdx.x = 3
```

```
int index = threadIdx.x + blockIdx.x * m
```

If we want to operate on the 21\textsuperscript{st} element of the array

```
int index = 5 + 2 \times 8 = 21
```

This can be generalized for grids with more dimensions than one
Vector Addition

1. Using 1 block and N threads
   __global__ void addVecs(int *c, int *a, int *b){
   c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
   }

2. Using N block and 1 thread per block
   __global__ void addVecs(int *c, int *a, int *b){
   c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
   }

3. Using n blocks and m threads per block (n*m=N)
   __global__ void addVecs(int *c, int *a, int *b){
   int index = threadIdx.x + blockIdx.x * blockDim.x;
   c[index] = a[index] + b[index];
   }
Call it: addVecs<<<n,m>>>(d_c,d_a,d_b);
Vector Addition

- Usually the vector size is not multiples of `blockDim.x`!!!
- One should be careful and not access memory beyond the array
- We can create 1 more block and from this block only threads with index which is smaller than the dimensions of the array should operate
Vector Addition

• Usually the vector size is not multiples of blockDim.x!!!
• One should be careful and not access memory beyond the array
• We can create 1 more block and from this block only threads with index which is smaller than the dimensions of the array should operate

We launch the kernel in a different way

```c
addVecs<<((N+m-1)/m,m)>>>(d_c, d_a, d_b, N);

__global__ void addVecs(int *c, int *a, int *b, int L){
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < L) // Rest of threads will be idle
        c[index] = a[index] + b[index];
}
```
Why one should combine threads & blocks

• It is important to create many thread blocks to keep SMs busy

• Why do we need to have more than one thread per block
  - Cons: Increase complexity
  - Pros:
    1. Can communicate using shared memory
    2. Utilize all CUDA cores inside a SM
    3. Warp scheduler can hide memory latency

• Blocks running completely in parallel and we cannot synchronize them
• Threads need synchronization
Vector dot product

Let's see an example of dot product using one block

```c
__global__ void dotVecs(float *x, float *y, float *r){
    float temp = x[threadIdx.x] * y[threadIdx.x];
    // TODO: Perform reduction
    // TODO: Store the result to *r
}
```

Where variable `temp` is stored?
Vector dot product

Let’s see an example of dot product using one block

```c
__global__ void dotVecs(float *x, float *y, float *r){
    float temp = x[threadIdx.x] * y[threadIdx.x];
    // TODO: Perform reduction
    // TODO: Store the result to *r
}
```

Where variable `temp` is stored?

- Scalar variables are automatically stored in registers by the compiler
- Register memory is the fastest but limited (Pascal: 256KB / SM)
- `temp` is local for each thread, therefore reduction is not possible
Vector dot product

Let’s see an example of dot product using one block

```c
__global__ void dotVecs(float *x, float *y, float *r) {
    float temp = x[threadIdx.x] * y[threadIdx.x];
    // TODO: Perform reduction
    // TODO: Store the result to *r
}
```

Where variable `temp` is stored?

- Scalar variables are automatically stored in registers by the compiler
- Register memory is the fastest but limited (Pascal: 256KB / SM)
- `temp` is local for each thread, therefore reduction is not possible

We need to use the shared memory to be able to perform reduction in each block
Vector dot product

#define N 1024
__global__ void dotVecs(float *x, float *y, float *r)
{
    __shared__ float s_tmp[N];
    float temp = x[threadIdx.x] * y[threadIdx.x];
    s_tmp[threadIdx.x] = temp; // store the multiplication to the shared memory

    // Thread 0 performs the reduction
    if(threadIdx.x == 0){
        float sum = 0;
        for(int i = 0 ; i < N ; i++) sum += s_tmp[i];
        *r = sum;
    }
}
#define N 1024
__global__ void dotVecs(float *x, float *y, float *r){
    __shared__ float s_tmp[N];
    float temp = x[threadIdx.x] * y[threadIdx.x];
    s_tmp[threadIdx.x] = temp;  // store the multiplication to the shared memory

    // Thread 0 performs the reduction
    if(threadIdx.x == 0){
        float sum = 0;
        for(int i = 0 ; i < N ; i++) sum += s_tmp[i];
        *r = sum;
    }
}
Vector dot product

Let’s take the scenario:

1. Thread 0 performs the multiplication and saves the result \( s_{tmp}[0] \)
2. Thread 0 performs the reduction \( \text{sum} += s_{tmp}[32] \)
3. But wait!! Did thread 32 already saved its value to \( s_{tmp}[32] \)
4. It is possible that thread 32 did not update the value yet because threads in different warps run asynchronously
Let's take the scenario:

1. Thread 0 performs the multiplication and saves the result `s_tmp[0]`
2. Thread 0 performs the reduction `sum += s_tmp[32]`
3. But wait!! Did thread 32 already saved its value to `s_tmp[32]`?
4. It is possible that thread 32 did not update the value yet because threads in different warps run asynchronously.

**Solution:** We have to impose synchronization barrier on the threads before the reduction.
Vector dot product

#define N 1024
__global__ void dotVecs(float *x, float *y, float *r){
  __shared__ float s_tmp[N];
  float temp = x[threadIdx.x] * y[threadIdx.x];
  s_tmp[threadIdx.x] = temp; // store the multiplication to the shared memory

  __syncthreads(); // synchronization barrier for the threads inside a block
  // Thread 0 performs the reduction
  if(threadIdx.x == 0){
    float sum = 0;
    for(int i = 0 ; i < N ; i++) sum += s_tmp[i];
    *r = sum;
  }
}
Vector dot product using more than one block

#define N 1024
#define N_THR 512  // we call this kernel with 2 blocks
__global__ void dotVecs(float *x, float *y, float *r){
    __shared__ float s_tmp[N_THR];
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    float temp = x[index] * y[index];
    s_tmp[threadIdx.x] = temp;  // store the multiplication to the shared memory
    __syncthreads();  // synchronization barrier for the threads inside a block
    // Thread 0 performs the reduction
    if(threadIdx.x == 0){  // One thread from each block will perform the reduction
        float sum = 0;
        for(int i = 0 ; i < N_THR ; i++) sum += s_tmp[i];
        *r += sum;  // r is initialized to zero before we call this kernel
    }
}
Vector dot product using more than one block

#define N 1024
#define N_THR 512  // we call this kernel with 2 blocks
__global__ void dotVecs(float *x, float *y, float *r)
{
  __shared__ float s_tmp[N_THR];
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  float temp = x[index] * y[index];
  s_tmp[threadIdx.x] = temp;  // store the multiplication to the shared memory

  __syncthreads();          // synchronization barrier for the threads inside a block
  // Thread 0 performs the reduction
  if(threadIdx.x == 0){    // One thread from each block will perform the reduction
    float sum = 0;
    for(int i = 0 ; i < N_THR ; i++) sum += s_tmp[i];
    *r += sum;  // r is initialized to zero before we call this kernel
  }
}

Do you see any problem here?
Vector dot product

- The previous code hides a potential race condition
- The 2 blocks running asynchronously
- The 2 thread blocks will try to update the same memory location
Vector dot product

- The previous code hides a potential race condition
- The 2 blocks running asynchronously
- The 2 thread blocks will try to update the same memory location

**Update process:**

(I) Read current value of \(*r\) in a temporary location
(II) Update the value in a temporary location
(III) Store the updated result back in \(*r\)
Vector dot product

- The previous code hides a potential race condition
- The 2 blocks running asynchronously
- The 2 thread blocks will try to update the same memory location

**Update process:**

(I) Read current value of *r in a temporary location
(II) Update the value in a temporary location
(III) Store the updated result back in *r

**Scenario:**

<table>
<thead>
<tr>
<th>Block</th>
<th>t=0</th>
<th>t=1</th>
<th>t=2</th>
<th>t=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reads *r → 0</td>
<td>Updates 0+4=4</td>
<td>Saves *r ← 4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Reads *r → 0</td>
<td>Updates 0+5=5</td>
<td>Saves *r ← 5</td>
</tr>
</tbody>
</table>
Vector dot product

- The previous code hides a potential race condition
- The 2 blocks running asynchronously
- The 2 thread blocks will try to update the same memory location

**Update process:**

(I) Read current value of *r in a temporary location
(II) Update the value in a temporary location
(III) Store the updated result back in *r

**Scenario:**

<table>
<thead>
<tr>
<th>Block</th>
<th>t=0</th>
<th>t=1</th>
<th>t=2</th>
<th>t=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reads *r → 0</td>
<td>Updates 0+4=4</td>
<td>Saves *r ← 4</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>Reads *r → 0</td>
<td>Updates 0+5=5</td>
<td>Saves *r ← 5</td>
</tr>
</tbody>
</table>

Result at *r after update process will be 5 instead of 9!!! Atomic function
Atomic Functions:

- Perform a read-modify-write operation
- On a 32- or 64-bit words
- The word can residing in device or shared memory
- Atomic operations are guaranteed to be performed without interference from other threads
Vector dot Product AtomicAdd

Atomic Functions:

- Perform a read-modify-write operation
- On a 32- or 64-bit words
- The word can residing in device or shared memory
- Atomic operations are guaranteed to be performed without interference from other threads

\[ \ast r += \text{sum} \quad \text{atomicAdd}(r, \text{sum}) \]

This guarantees that the result is free of race conditions.
Dynamic shared memory allocation

In our previous code the size of the shared memory was hard-coded

__shared__ float s_tmp[N];

where N will be substituted by the preprocessor
Dynamic shared memory allocation

In our previous code the size of the shared memory was hard-coded

```c
__shared__ float s_tmp[N];
```

where N will be substituted by the preprocessor

If we want to **dynamically** allocate the size of the shared memory

```c
extern __shared__ float s_tmp[];
```

And we call the kernel using an additional argument:

```c
dotVecs<<<(N+m-1)/m,m,m*sizeof(float)>>>(d_x, d_y, d_r, N);
```
Create CUDA API events

In case that we want to **measure the performance** of our code we use CUDA events. This cannot be done using CPU timings because GPU can execute asynchronously with respect to CPU.
Create CUDA API events

In case that we want to **measure the performance** of our code we use CUDA events. This cannot be done using CPU timings because GPU can execute asynchronously with respect to CPU.

```c
cudaEvent_t start, stop; // define start and stop event variable
cudaEventCreate(&start); cudaEventCreate(&stop); // initializes the event variables
cudaEventRecord(start,0); // it puts the timestamp for event start

// cudaMalloc ...
// cudaMemcpy ...
// execute Kernels ...

cudaEventRecord(stop,0); // it puts the timestamp for the event stop
cudaEventSynchronize(stop); // very important, it instructs CPU to synch. on stop event
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
printf("Elapsed Time is \%f ms\n",elapsedTime)
cudaEventDestroy(start); cudaEventDestroy(stop);
```
Create CUDA API events

In case that we want to **measure the performance** of our code we use CUDA events. This cannot be done using CPU timings because GPU can execute asynchronously with respect to CPU.

```c
cudaEvent_t start, stop; // define start and stop event variable
cudaEventCreate(&start); cudaEventCreate(&stop); // initializes the event variables
cudaEventRecord(start,0); // it puts the timestamp for event start

// cudaMalloc ...
// cudaMemcpy ...
// execute Kernels ...

// cudaEventRecord(stop,0); // it puts the timestamp for the event stop
cudaEventSynchronize(stop); // very important, it instructs CPU to synch. on stop event

float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
printf("Elapsed Time is %f ms\n",elapsedTime)
cudaEventDestroy(start); cudaEventDestroy(stop);
```

What the second argument means here?
CUDA Streams

In the previous example “0” was used to denote the default CUDA stream. But what CUDA streams are??
CUDA Streams

In the previous example “0” was used to denote the default CUDA stream

But what CUDA streams are??

• CUDA streams can help in accelerating our application
• Represent a queue of GPU operations
• We can add operations such as, memory copies, kernel launches and start stop of events on CUDA stream
CUDA Streams

In the previous example “0” was used to denote the default CUDA stream

But what CUDA streams are??

- CUDA streams can help in accelerating our application
- Represent a queue of GPU operations
- We can add operations such as, memory copies, kernel launches and start stop of events on CUDA stream

GPUs allow for overlapping of computation and communication ➡ But this needs asynchronous copy between host and device

cudaHostAlloc() : For asynchronous copies we need host memory which is page-locked

To deallocate host memory allocated by cudaHostAlloc() one has to use cudaFreeHost() instead of free()
**CUDA Streams (add vectors)**

<table>
<thead>
<tr>
<th>Stream 0</th>
<th>Stream 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpy X to GPU</td>
<td>cpy X to GPU</td>
</tr>
<tr>
<td>cpy Y to GPU</td>
<td>cpy Y to GPU</td>
</tr>
<tr>
<td>call Kernel</td>
<td>call Kernel</td>
</tr>
<tr>
<td>cpy Z from GPU</td>
<td>cpy Z from GPU</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Kernel execution is overlapped with copying
Vector Addition using CUDA streams

```c
#define N (20*1024*1024)
#define CHUNK_SIZE (1024*1024)

__global__ void add(int *z, int *x, int *y){
  // each thread will add one element...
}

int main(){
  int *h_x, *h_y, *h_z;
  int *d_x0, *d_y0, *d_z0; // for stream 0
  int *d_x1, *d_y1, *d_z1; // for stream 1

  cudaEvent_t start, stop;
  float elapsedTime;

  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaMemcpy(h_x, d_x0, N * sizeof(int), cudaMemcpyDeviceToHost);
  cudaMemcpy(h_y, d_y0, N * sizeof(int), cudaMemcpyDeviceToHost);
  cudaMemcpy(h_z, d_z0, N * sizeof(int), cudaMemcpyDeviceToHost);

  // Allocate device memory
  cudaMemcpy(d_x1, d_x0, CHUNK_SIZE * sizeof(int), cudaMemcpyDeviceToDevice);
  cudaMemcpy(d_y1, d_y0, CHUNK_SIZE * sizeof(int), cudaMemcpyDeviceToDevice);
  cudaMemcpy(d_z1, d_z0, CHUNK_SIZE * sizeof(int), cudaMemcpyDeviceToDevice);

  // Allocate stream
  cudaStream_t stream0, stream1;
  cudaStreamCreate(&stream0);
  cudaStreamCreate(&stream1);

  // Allocate page-locked host memory
  cudaMemcpy(h_x, d_x0, N * sizeof(int), cudaMemcpyDeviceToHost);
  cudaMemcpy(h_y, d_y0, N * sizeof(int), cudaMemcpyDeviceToHost);
  cudaMemcpy(h_z, d_z0, N * sizeof(int), cudaMemcpyDeviceToHost);

  // Initialize vectors with random numbers
  random(h_x, N);
  random(h_y, N);
```
for(int i = 0; i < N ; i += 2*CHUNK_SIZE){
    // operations on stream0
    cudaMemcpyAsync(d_x0, h_x+i, CHUNK_SIZE*sizeof(int), cudaMemcpyHostToDevice,stream0);
    cudaMemcpyAsync(d_y0, h_y+i, CHUNK_SIZE*sizeof(int), cudaMemcpyHostToDevice,stream0);
    addVecs<<<CHUNK_SIZE,CHUNK_SIZE/256, 0, stream0>>>(d_z0, d_x0, d_y0);
    cudaMemcpyAsync(h_z+i, d_z0, CHUNK_SIZE*sizeof(int), cudaMemcpyDeviceToHost,stream0);
    // operations on stream1
    cudaMemcpyAsync(d_x1, h_x+i+CHUNK_SIZE, CHUNK_SIZE*sizeof(int), cudaMemcpyHostToDevice,stream1);
    cudaMemcpyAsync(d_y1, h_y+i+CHUNK_SIZE, CHUNK_SIZE*sizeof(int), cudaMemcpyHostToDevice,stream1);
    addVecs<<<CHUNK_SIZE,CHUNK_SIZE/256, 0, stream1>>>(d_z1, d_x1, d_y1);
    cudaMemcpyAsync(h_z+i+CHUNK_SIZE, d_z1, CHUNK_SIZE*sizeof(int), cudaMemcpyDeviceToHost,stream1);
}
// we need to sync both streams
.cudaStreamSynchronize(stream0);
.cudaStreamSynchronize(stream1);
.cudaEventRecord(stop,0);
.cudaEventSynchronize(stop);
.cudaEventElapsedTime(&elapsedTime, start, stop);
.printf("Elapsed Time is %f ms \n",elapsedTime);
.cudaFreeHost(h_x); cudaFreeHost(h_y); cudaFreeHost(h_z);
.cudaFree(d_x0); cudaFree(d_y0); cudaFree(d_z0); cudaFree(d_x1); cudaFree(d_y1); cudaFree(d_z1);
.cudaStreamDestroy(stream0); cudaStreamDestroy(stream1); cudaEventDestroy(start); cudaEventDestroy(stop);
return 0;
Vector Addition using Zero-copy memory

Do CUDA kernels need data to be located on Device memory to access it?
Vector Addition using Zero-copy memory

Do CUDA kernels need data to be located on Device memory to access it? No!!
Device can access data on host memory directly
Vector Addition using Zero-copy memory

Do CUDA kernels need data to be located on Device memory to access it? No!!

Device can access data on host memory directly

```c
int main(){
    // check if the device supports this (true for new architectures)
    cudaSetDeviceFlags(cudaDeviceMapHost);
    // declare host and device pointers
    // Allocate page-locked host memory
    cudaHostAlloc((void**)&h_x, N*sizeof(int), cudaHostAllocMapped);
    cudaHostAlloc((void**)&h_y, N*sizeof(int), cudaHostAllocMapped);
    cudaHostAlloc((void**)&h_z, N*sizeof(int), cudaHostAllocMapped);

    // Host memory has different virtual memory than device
    cudaHostGetDevicePointer(&d_x, h_x, 0);
    cudaHostGetDevicePointer(&d_y, h_y, 0);
    cudaHostGetDevicePointer(&d_z, h_z, 0);
    // cudaSetDeviceFlags(cudaDeviceMapHost) is important otherwise cudaMemcpy
    // There is no need for cudaMemcpy
    addVecs<<<...>>>(d_z, d_x, d_y);
    // it is important to wait until the kernel finishes before we access mapped host data
    cudaDeviceSynchronize();
    // free memory ...
}
```
Catch errors in CUDA

- Up to now we haven’t seen a case where a CUDA API function returns an error
- What it will happen if a cudaMalloc returns “out of memory”?

```c
#define CudaSafeCall( err ) __cudaSafeCall( err, __FILE__, __LINE__ )

inline void __cudaSafeCall( cudaError_t err, const char *file, const int line )
{
    if ( cudaSuccess != err )
    {
        fprintf( stderr, "cudaSafeCall() failed at %s:%i : %s\n", file, line, cudaGetErrorString( err ) );
        exit( -1 );
    }
    return;
}

CudaSafeCall(cudaMalloc(…));
CudaSafeCall(cudaMemcpy(…));
```
# What about error from CUDA kernels

```c
#define CudaCheckError() __cudaCheckError(__FILE__, __LINE__ )
inline void __cudaCheckError( const char *file, const int line )
{
    cudaError_t err = cudaGetLastError();
    if ( cudaSuccess != err )
    {
        fprintf( stderr, "cudaCheckError() failed at %s:%i : %s\n", file, line, cudaGetErrorString( err ));
        exit( -1 );
    }
    // More careful checking. However, this will affect performance.
    // Cuda Kernels are async. and one should use cudaDeviceSynchronize
    // Comment away if needed.
    err = cudaDeviceSynchronize();
    if( cudaSuccess != err )
    {
        fprintf( stderr, "cudaCheckError() with sync failed at %s:%i : %s\n", file, line, cudaGetErrorString( err ) );
        exit( -1 );
    }
    return;
}
```

We put CudaCheckError() at the point where we want to check if there is an error.
Device Management

Check how many GPUs we have available:
• cudaGetDeviceCount(int *count)

Choose the device that we want to use (default is 0):
• cudaSetDevice(int device)

Check which device is using the active CPU thread which binds the device:
• cudaGetDevice(int *device)

Check the properties of the device:
• cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
  - size_t totalGlobalMem;
  - size_t sharedMemPerBlock;
  - int warpSize;
  - int deviceOverlap;
  - ...

A single host thread can manage multiple devices: cudaSetDevice(i)
Useful tools

- **CUDA-GDB**: CUDA debugger which allows debugging of both CPU and GPU portions of your code
  - Uses conditional breakpoints to identify and correct errors in CUDA code
  - Identify memory access violations
  - Selectively assert in CUDA code
  - And many other features
Useful tools

- **CUDA-GDB**: CUDA debugger which allows debugging of both CPU and GPU portions of your code
  - Uses conditional breakpoints to identify and correct errors in CUDA code
  - Identify memory access violations
  - Selectively assert in CUDA code
  - And many other features

- **NVIDIA Visual Profiler**: The NVIDIA Visual Profiler is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications.
Topics we discussed:

- Write and launch CUDA kernels
  - `__global__`, `__device__`, `blockIdx.x`, `threadIdx.x`, `blockDim.x`, `<<<>>>`
- Manage GPU memory and CPU memory
  - `cudaMalloc()`, `cudaMemcpy()`, `cudaFree()`, `cudaHostAlloc()`, `cudaHostFree()`
- Communication
  - `__shared__` (static, dynamic)
- Synchronization
  - `__syncthreads()`
  - `cudaDeviceSynchronize()`
- Overlapping copying and execution
  - `cudaMemcpyAsync`
  - CUDA streams

Topics we skipped:

- Textures objects, banks conflicts, constant memory, …