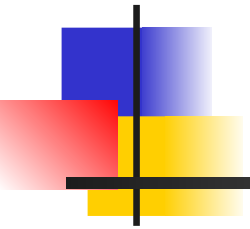


CS476/576: Programming Models for Emerging Platforms

GPU Programming for Non-Graphics

(CUDA)

CUDA

- A programming model specifically targeting GPGPU programming
  - Developed by Nvidia
  - A C-like language
- Traits
  - Host-side programming (what previously was XNA/ OpenGL/DirectX) and device-side programming (what previously was Cg/HLSL) are unified in one language
  - Shared memory between different GPU cores is supported
  - Simplifications on syntax, such as kernel (shader) invocations, no semantic bindings, etc

CUDA-Enabled Hardware

- Often a 2-level hierarchy:
  - Stream Multi-processors (SMs): the unit of hardware chip (analogous to a group of, say, fragment processors)
  - Cores: parallel execution unit on each SM (analogous to a fragment processor)
- Example: Tesla C2050 has 14 SMs and each has 32 cores. So in total it has 14 *32 = 448 cores
- Be careful about terminology of “cores”: if you read “CUDA Programming Guide”, “cores” there mean SMs (as confusing as it gets!)

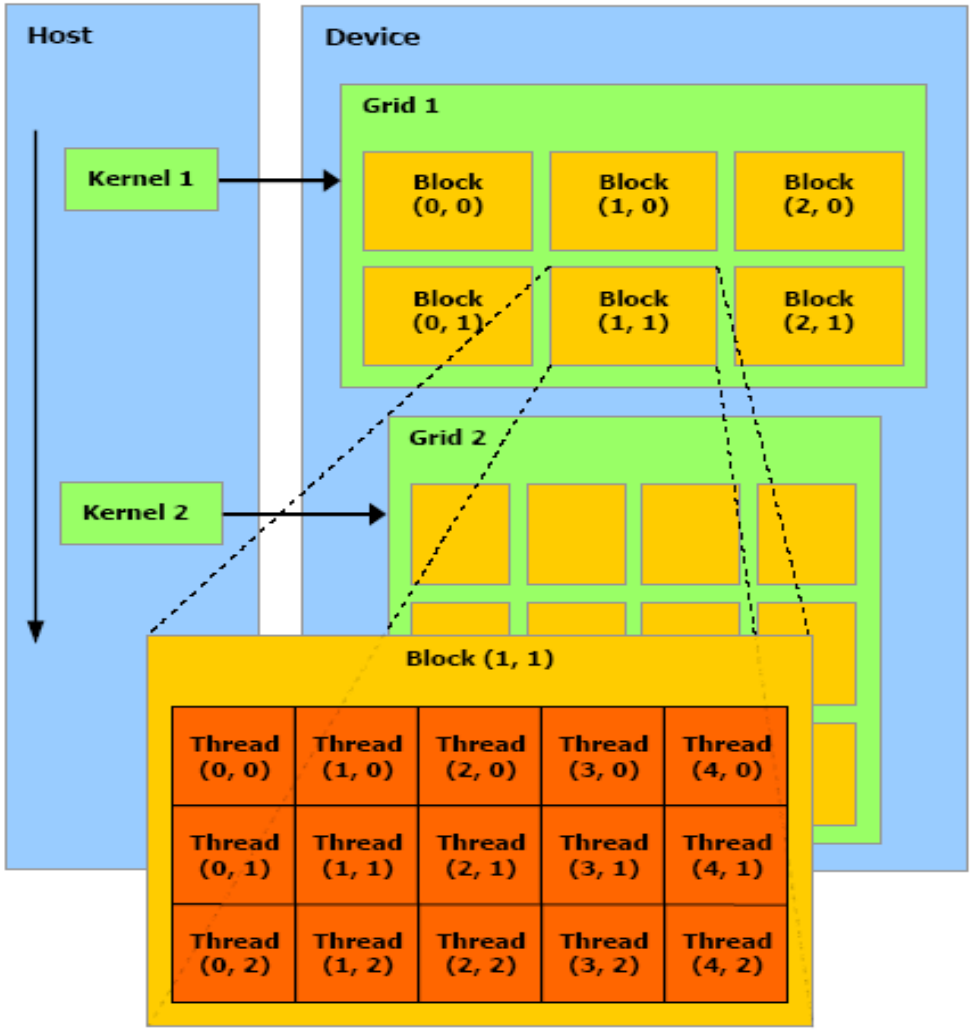
Shared Memory: A Hardware View

- Each SM is associated with an on-chip low-latency memory area
  - For Tesla C2050, it is 64KB
- The memory is shared by all threads running on the same SM
- Usually, this memory area can be configured as a combination of programmer-controlled shared memory and data cache

Thread Model

- Each GPU core executes an instance of the kernel function, called a thread.
- The batch of threads are organized as a grid of thread blocks
  - The grid has many blocks
  - Each block has many threads
- Blocks are mapped to SMs whereas threads are mapped to cores

Hardware Model



The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks

Host Program

- Very simple kernel invocation syntax
- It's still up to the programmer to explicitly coping data from CPU to GPU and back

```
int main(int argc, char *argv[]){

  … // variable declaration omitted

  h_A = (float *)malloc(DATA_SZ);
  … // similar for h_B

  cudaMalloc((void **)&d_A, DATA_SZ);
  … // similar for d_B, d_C

  cudaMemcpy(d_A, h_A, DATA_SZ,
    cudaMemcpyHostToDevice);
  … // similar for d_B

  ProdGPU<<<BLOCK_N, THREAD_N>>>(d_C, d_A,
    d_B);
  …

  cudaMemcpy(h_C_GPU, d_C, RESULT_SZ,
    cudaMemcpyDeviceToHost);
  …

  cudaFree(d_A);
  free(h_A);
  …


}
```

A Two-Tiered Logical Model

- Threads in the same thread block runs on the same SM, which is equipped with on-chip shared memory
  - Fast communications
  - Synchronization possible
- Both blocks and threads can indexed as one-, two- or three-dimensional arrays

CUDA Program Elements

- Function type qualifiers to specify execution on host or device
- Variable type qualifiers to specify the memory location on the device
- A new directive to specify how to execute a kernel on the device
- Four built-in variables that specify the grid and block dimensions and the block and thread indices

CUDA Basics

- Function type qualifiers (default is __host__)

__device__

- Executed on the device
- Callable from the device only.

__global__

- Executed on the device,
- Callable from the host only.

__host__

- Executed on the host,
- Callable from the host only.

CUDA Basics

- Variable Type Qualifiers

  __device__
  - Resides in global memory space,
  - Has the lifetime of an application,
  - Is accessible from all the threads within the grid and from the host through some library APIs.

  __constant__
  - Resides in constant memory space,
  - Has the lifetime of an application,
  - Is accessible from all the threads within the grid and from the host through some library APIs.

  __shared__
  - Resides in the shared memory space of a thread block,
  - Has the lifetime of the block,
  - Is only accessible from all the threads within the block.
- If not declared, the variables defined in device functions (or formal parameters) is an automatic variable for GPU registers or local memory
- __device__ and __constant__ are only used for file-scoped variable declarations

CUDA Basics

- Kernel Invocation

  ```
  __global__ void Func(float* parameter);
  ```

  must be called like this:

  ```
  Func<<< Dg, Db >>>(parameter);
  ```

- **Dg is the number of the blocks, and Db is the number of threads per block.**

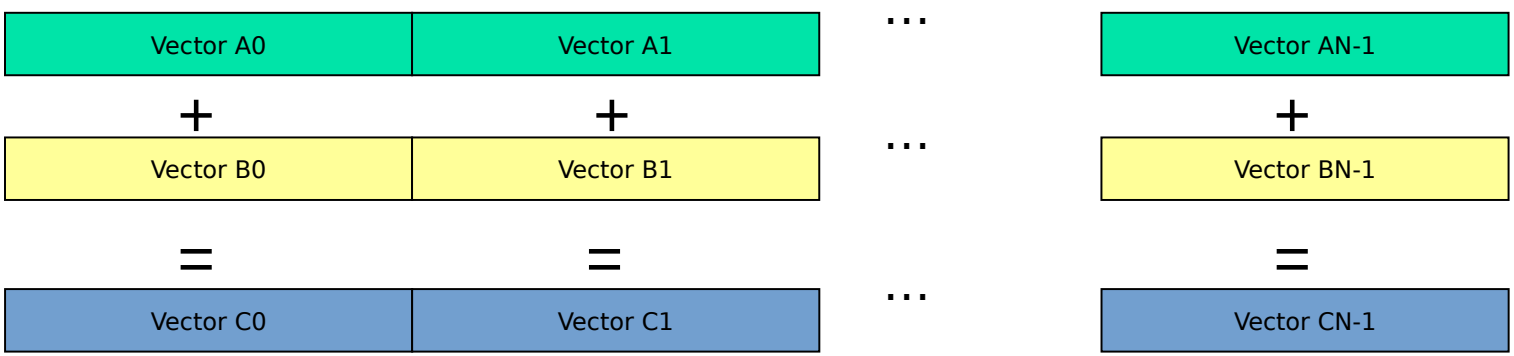
- **Dg and Db do not have to be integers. They can also be tuples. For example, if Dg is (3, 2), then it is to define a program execution on 6 blocks. The only difference is now you can refer to each block as (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1) instead of 1, 2, 3, 4, 5. 6.**

CUDA Basics

- Built-in Variables
  - **`gridDim`** is the number of blocks
  - **`blockIdx`** is the index of “my” block
  - **`blockDim`** is the number of threads in a block
  - **`threadIdx`** is the index of “my” thread

Example: Vector Addition

- Calculate the scalar product of
  - 32 vector pairs
  - 4096 elements each



CPU Version

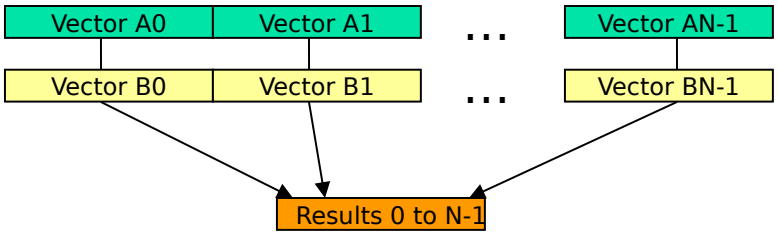
```
int main(void) {
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the CPU
  add(N, x, y);
}

void add(int n, float *x, float *y) {
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}
```

A Simplified Kernel Function

```
__global__ void ProdGPU(float *d_C, float *d_A, float *d_B){
    __shared__ float t[THREAD_N];
    const int vec_n=blockIdx.x;
    const int I = threadIdx.x;
    int base = ELEMENT_N * vec_n;
    t[I] = d_A[base + I] * d_B[base + I];
    __syncthreads();
    for(int stride = THREAD_N / 2; stride > 0; stride /= 2){
        if(I < stride) t[I] += t[stride + I];
          __syncthreads();
      }
    if(I == 0) d_C[vec_n] = t[0];
}
```

Example: Scalar Product

- Calculate the scalar product of
  - 32 vector pairs
  - 4096 elements each



The data will be handed to the device as two data arrays and the results will be saved in a result array

BLOCK_N = 32
THREAD_N = 4096

Host Program

```
int main(int argc, char *argv[]){
  …
  h_A = (float *)malloc(DATA_SZ);
  … // similar for h_B
  cudaMalloc((void **)&d_A, DATA_SZ);
  … // similar for d_B, d_C
  cudaMemcpy(d_A, h_A, DATA_SZ, cudaMemcpyHostToDevice);
  … // similar for d_B

  ProdGPU<<<BLOCK_N, THREAD_N>>>(d_C, d_A, d_B);


  …
  cudaMemcpy(h_C_GPU, d_C, RESULT_SZ, cudaMemcpyDeviceToHost);
  …
  cudaFree(d_A);
  free(h_A);
  …
}
```

Parameters:

d_C: pointer to result array

d_A, d_B pointers to input data

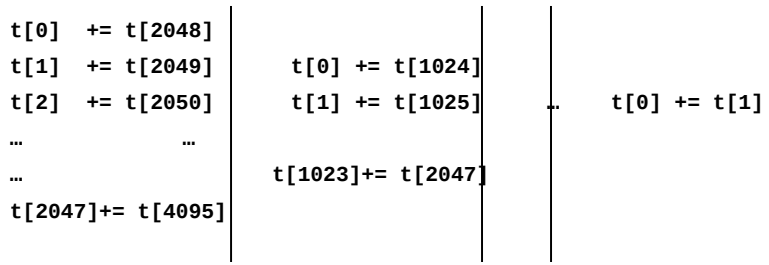
A Simplified Kernel Function

```
__global__ void ProdGPU(float *d_C, float *d_A, float *d_B){
    __shared__ float t[THREAD_N];
    const int vec_n=blockIdx.x;
    const int I = threadIdx.x;
    int base = ELEMENT_N * vec_n;
    t[I] = d_A[base + I] * d_B[base + I];
    __syncthreads();
    for(int stride = THREAD_N / 2; stride > 0; stride /= 2){
        if(I < stride) t[I] += t[stride + I];
          __syncthreads();
      }
    if(I == 0) d_C[vec_n] = t[0];
}
```

Example: Scalar Product

The Kernel Function

- Calculate the result for each vector through “parallel reduction”



- Save the partial result

A Simplified Kernel Function

```
__global__ void ProdGPU(float *d_C, float *d_A, float *d_B){
    __shared__ float t[THREAD_N];
    const int vec_n=blockIdx.x;
    const int I = threadIdx.x;
    int base = ELEMENT_N * vec_n;
    t[I] = d_A[base + I] * d_B[base + I];
    __syncthreads();
    for(int stride = THREAD_N / 2; stride > 0; stride /= 2){
        if(I < stride) t[I] += t[stride + I];
          __syncthreads();
      }
    if(I == 0) d_C[vec_n] = t[0];
}
```

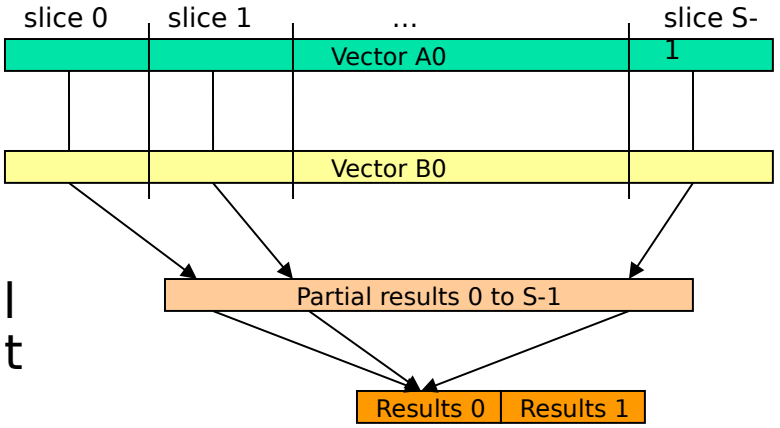
Example: Scalar Product

The __syncthreads() function guarantees that all kernel instances are “in sync”, i.e. it is a barrier that can be crossed only when all kernel instances reach the function.

- Why? Isn’t CUDA SIMD?

Efficient Scalar Product

- An efficient way to run that on the device is to organize the calculation in
  - A grid of 32 blocks
  - With 256 threads per block
- This gives us 4096/256 = 16 slices per vector



Each product of a vector pair An, Bn will be calculated in slices, which will be added up to obtain the final result

BLOCK_N = 32
THREAD_N = 256

More Efficient Kernel Function

```
__global__ void ProdGPU(float *d_C, float *d_A, float *d_B){
  __shared__ float t[THREAD_N];
  __shared__ float r[SLICE_N];
  const int I = threadIdx.x;
    int base = ELEMENT_N * vec_n;
    for(int slice = 0; slice < SLICE_N; slice++, base += THREAD_N)     {
      t[I] = d_A[base + I] * d_B[base + I];
      __syncthreads();
      for(int stride = THREAD_N / 2; stride > 0; stride /= 2){
        if(I < stride) t[I] += t[stride + I];
          __syncthreads();
      }
      if(I == 0) r[slice] = t[0];
    }
    for(int stride = SLICE_N / 2; stride > 0; stride /= 2){
      if(I < stride) r[I] += r[stride + I];
      __syncthreads();
    }
    if(I == 0) d_C[vec_n] = r[0];

}
```

More Efficient Kernel Function

```
__global__ void ProdGPU(float *d_C, float *d_A, float *d_B){
  __shared__ float t[THREAD_N];
  __shared__ float r[SLICE_N];
  const int I = threadIdx.x;
    int base = ELEMENT_N * vec_n;
    for(int slice = 0; slice < SLICE_N; slice++, base +=
 THREAD_N)     {
      t[I] = d_A[base + I] * d_B[base + I];
      __syncthreads();
      for(int stride = THREAD_N / 2; stride > 0; stride /= 2){

        if(I < stride) t[I] += t[stride + I];
          __syncthreads();
      }
      if(I == 0) r[slice] = t[0];
    }
    for(int stride = SLICE_N / 2; stride > 0; stride /= 2){
      if(I < stride) r[I] += r[stride + I];
      __syncthreads();
    }
    if(I == 0) d_C[vec_n] = r[0];

}
```

Complete Kernel Function

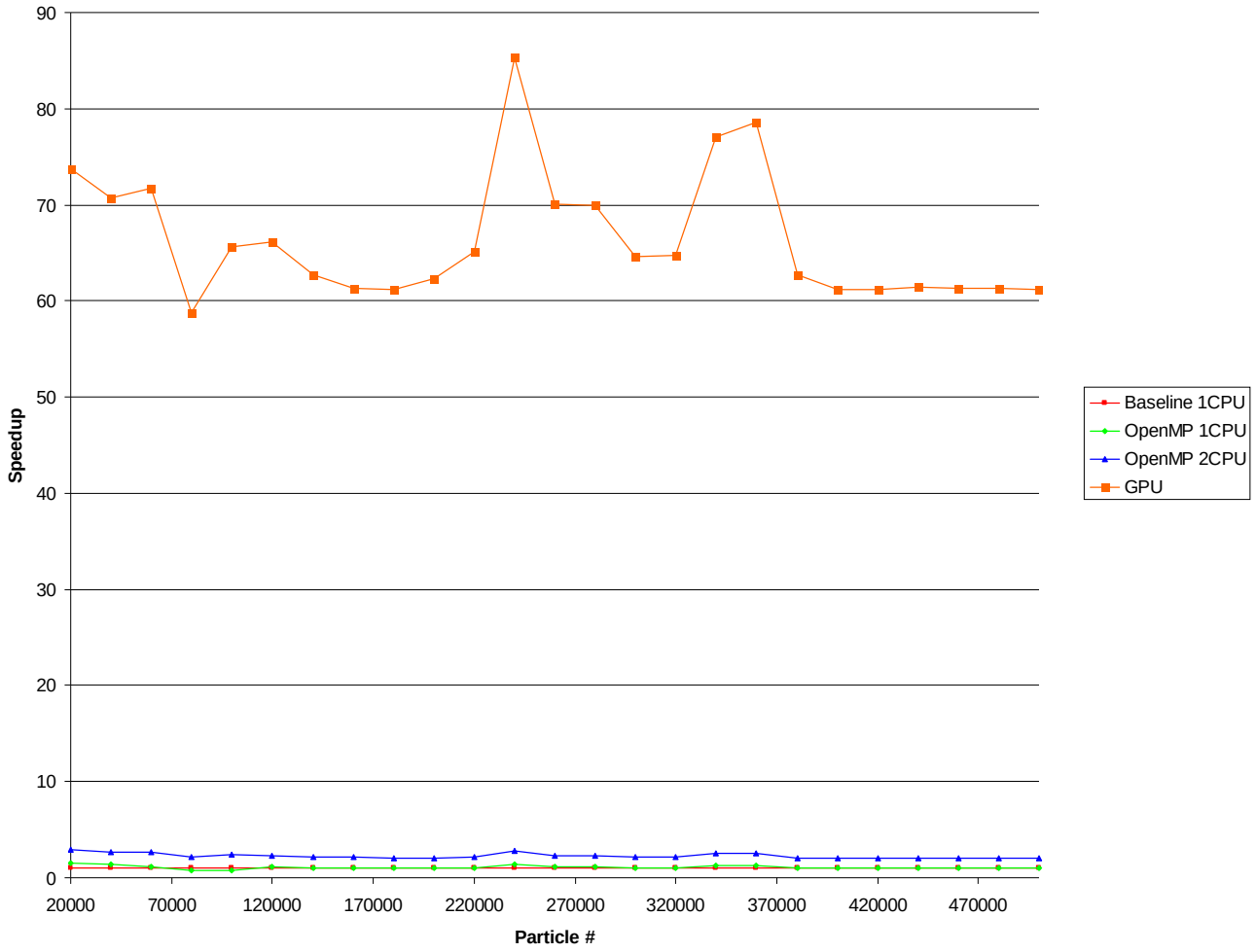
```
__global__ void ProdGPU(float *d_C, float *d_A, float *d_B){
  __shared__ float t[THREAD_N];
  __shared__ float r[SLICE_N];
  const int I = threadIdx.x;
  for(int vec_n=blockIdx.x; vec_n<VECTOR_N; vec_n+=gridDim.x){
    int base = ELEMENT_N * vec_n;
    for(int slice = 0; slice < SLICE_N; slice++, base +=
   THREAD_N)     {
      t[I] = d_A[base + I] * d_B[base + I];
      __syncthreads();
      for(int stride = THREAD_N / 2; stride > 0; stride /= 2){

        if(I < stride) t[I] += t[stride + I];
          __syncthreads();
      }
      if(I == 0) r[slice] = t[0];
    }
    for(int stride = SLICE_N / 2; stride > 0; stride /= 2){
      if(I < stride) r[I] += r[stride + I];
      __syncthreads();
    }
    if(I == 0) d_C[vec_n] = r[0];
  }
}
```

consider the general case where there are more vectors than blocks

A Common Programming Strategy: Tiling

- Partition data into subsets that fit into shared memory
- Handle each data subset with one thread block:
  - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
  - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
  - Copying results from shared memory to global memory

CPU/GPU Comparison (N Body)



CPU/GPU Comparison

- GPU Baseline speedup is approximately 60x
- For 500,000 particles that is a reduction in calculation time from 33 minutes to 33 seconds!

Acknowledgments

- Xiaoming Li, Nvidia CUDA basics
- ECE498AL, University of Illinois, Urbana Champaign