GPU programming: CUDA basics

Sylvain Collange
Inria Rennes – Bretagne Atlantique
sylvain.collange@inria.fr
This lecture: CUDA programming

- We have seen some GPU architecture

- Now how to program it?
Outline

- GPU programming environments
- CUDA basics
  - Host side
  - Device side: threads, blocks, grids
- Expressing parallelism
  - Vector add example
- Managing communications
  - Parallel reduction example
GPU development environments

For general-purpose programming (not graphics)

- Multiple toolkits
  - NVIDIA CUDA
  - Khronos OpenCL
  - Microsoft DirectCompute
  - Google RenderScript
- Mostly syntactical variations
  - Underlying principles are the same
- In this course, focus on NVIDIA CUDA
Higher-level programming

- Directive-based
  - OpenACC
  - OpenMP 4.0

- Language extensions / libraries
  - Microsoft C++ AMP
  - Intel Cilk+
  - NVIDIA Thrust, CUB

- Languages
  - Intel ISPC

... 

- Most corporations agree we need common standards...
  - But only if their own product becomes the standard!
Outline

- GPU programming environments
- CUDA basics
  - Host side
  - Device side: threads, blocks, grids
- Expressing parallelism
  - Vector add example
- Managing communications
  - Parallel reduction example
- Re-using data
  - Matrix multiplication example
Hello World in CUDA

- CPU “host” code + GPU “device” code

```c
__global__ void hello() {
}

int main() {
    hello<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Device code

Host code
Compiling a CUDA program

- Executable contains both host and device code
  - Device code in PTX and/or native
  - PTX can be recompiled on the fly (e.g. old program on new GPU)
- NVIDIA's compiler driver takes care of the process:

  `nvcc -o hello hello.cu`
**Control flow**

- Program running on CPUs
- Submit work to the GPU through the GPU driver
- Commands execute asynchronously
External memory: discrete GPU

Classical CPU-GPU model

- Split memory spaces
- Highest bandwidth from GPU memory
- Transfers to main memory are slower

Ex: Intel Core i7 4770, Nvidia GeForce GTX 780
External memory: embedded GPU

Most GPUs today

- Same memory
- May support coherent memory
  - GPU can read directly from CPU caches
- More contention on external memory
Data flow

- Main program runs on the host
  - Manages memory transfers
  - Initiate work on GPU
- Typical flow
Data flow

- Main program runs on the host
  - Manages memory transfers
  - Initiate work on GPU

- Typical flow
  - 1. Allocate GPU memory
Data flow

- Main program runs on the host
  - Manages memory transfers
  - Initiate work on GPU

- Typical flow
  - 1. Allocate GPU memory
  - 2. Copy inputs from CPU mem to GPU memory
Data flow

- Main program runs on the host
  - Manages memory transfers
  - Initiate work on GPU

- Typical flow
  - 1. Allocate GPU memory
  - 2. Copy inputs from CPU mem to GPU memory
  - 3. Run computation on GPU
Data flow

- Main program runs on the host
  - Manages memory transfers
  - Initiate work on GPU
- Typical flow
  - 1. Allocate GPU memory
  - 2. Copy inputs from CPU mem to GPU memory
  - 3. Run computation on GPU
  - 4. Copy back results to CPU memory
Example: $a + b$

- Our Hello World example did not involve the GPU
- Let's add up 2 numbers on the GPU
- Start from host code

```c
int main()
{
    float ab[] = {1515, 159};  // Inputs, in host mem
    float c;
    // c = ab[0] + ab[1];
    printf("c = \%f\n", c);
}
```

vectorAdd example: cuda/samples/0_Simple/vectorAdd
Step 1: allocate GPU memory

```c
int main()
{
    float ab[] = {1515, 159};   // Inputs, in host mem

    // Allocate GPU memory
    float *d_AB, *d_C;
    cudaMalloc((void **)&d_AB, 2*sizeof(float));
    cudaMalloc((void **)&d_C, sizeof(float));

    // Free GPU memory
    cudaFree(d_AB);
    cudaFree(d_C);
}
```

- Allocate space for a, b and c in GPU memory
- At the end, free memory
int main()
{
    float ab[] = {1515, 159}; // Inputs, CPU mem

    // Allocate GPU memory
    float *d_AB, *d_C;
    cudaMalloc((void **)&d_AB, 2*sizeof(float));
    cudaMalloc((void **)&d_C, sizeof(float));

    // Copy from CPU mem to GPU mem
    cudaMemcpy(d_AB, &ab, 2*sizeof(float), cudaMemcpyHostToDevice);

    // Copy results back to CPU mem
    cudaMemcpy(&c, d_C, sizeof(float), cudaMemcpyDeviceToHost);
    printf("c = %f\n", c);

    // Free GPU memory
    cudaFree(d_AB);
    cudaFree(d_C);
}
Step 3: launch kernel

```c
__global__ void addOnGPU(float * ab, float * c)
{
    *c = ab[0] + ab[1];
}
```

```c
int main()
{
    float ab[] = {1515, 159};    // Inputs, CPU mem
    // Allocate GPU memory
    float *d_AB, *d_C;
    cudaMalloc((void **)&d_AB, 2*sizeof(float));
    cudaMalloc((void **)&d_C, sizeof(float));
    // Copy from CPU mem to GPU mem
    cudaMemcpy(d_AB, &a, 2*sizeof(float), cudaMemcpyHostToDevice);
    float c; // Result on CPU
    // Copy results back to CPU mem
    cudaMemcpy(&c, d_C, sizeof(float), cudaMemcpyDeviceToHost);
    printf("c = %f\n", c);
    // Free GPU memory
    cudaFree(d_AB);
    cudaFree(d_C);
}
```

- **Kernel is a function prefixed by `__global__`**
  - Runs on GPU
- **Invoked from CPU code with `<<<>>>` syntax**

Note: we could have passed `a` and `b` directly as kernel parameters

What is inside the `<<<>>>`?
Asynchronous execution

- By default, GPU calls are *asynchronous*
  - Returns immediately to CPU code
  - GPU commands are still executed in-order: queuing
- Some commands are synchronous by default
  - cudaMemcpy(..., cudaMemcpyDeviceToHost)
  - Use cudaMemcpyAsync for asynchronous version
- Keep it in mind when checking for errors!
  - Error returned by a command may be caused by an earlier command
- To force synchronization: cuThreadSynchronize()
Outline

- GPU programming environments
- CUDA basics
  - Host side
  - Device side: threads, blocks, grids
- Expressing parallelism
  - Vector add example
- Managing communications
  - Parallel reduction example
Granularity of a GPU task

Results from last Thursday lab work

- Total latency of transfer, compute, transfer back: ~5 µs
  - CPU-GPU transfer latency: 0.3 µs
  - GPU kernel call: ~4 µs
- CPU performance: 100 Gflop/s → how many flops in 5 µs?
Granularity of a GPU task

Results from last Thursday lab work

- Total latency of transfer, compute, transfer back: ~5 µs
  - CPU-GPU transfer latency: 0.3 µs
  - GPU kernel call: ~4 µs
- CPU performance: 100 Gflop/s → 500 000 flops in 5 µs
- For < 500k operations, computing on CPU will be always faster!
  - Millions of operations needed to amortize data transfer time
  - Only worth offloading large parallel tasks the GPU
GPU physical organization

- Thread
- Warp
- Execution units
- Registers
- Shared memory
- L1 cache

SM 1

To L2 cache / external memory

SM 2
Workload: logical organization

- A kernel is launched on a grid: `my_kernel<<<blocks, threads>>>(...)`
- Two nested levels
  - Blocks
  - Threads
Outer level: grid of blocks

- *Blocks* or *Concurrent Thread Arrays (CTAs)*
- **No communication** between blocks of the same grid
- No practical limit on the number of blocks
Inner level: threads

- Blocks contain threads
- All threads in a block
  - Run on the same SM: they can **communicate**
  - Run in parallel: they can **synchronize**
- Constraints
  - Max number of threads/block (512 or 1024 depending on arch)
  - Recommended: at least 64 threads for good performance
  - Recommended: multiple of the warp size
Multi-BSP model: recap

- Modern parallel platforms are hierarchical
  - Threads $\in$ cores $\in$ nodes...
  - Remember the memory wall, the speed of light
- Multi-BSP: BSP generalization with multiple nested levels

- Higher level: more expensive synchronization
Multi-BSP and CUDA

Minor difference: BSP is based on message passing, CUDA on shared memory
Mapping blocks to hardware resources

- SM resources are partitioned across blocks

- SM 1
  - Warp
  - Execution units
  - Registers
  - Shared memory
  - L1 cache
  - To L2 cache / external memory

- SM 2
  - Block 3
  - Block 4

- Block 1
  - Block 2

- ...
Block scheduling

- Blocks may
  - Run serially or in parallel
  - Run on the same or different SM
  - Run in order or out of order
- Should not assume anything on execution order of blocks
Block scheduling

- Blocks may
  - Run serially or in parallel
  - Run on the same or different SM
  - Run in order or out of order
- Should not assume anything on execution order of blocks
Block scheduling

- Blocks may
  - Run serially or in parallel
  - Run on the same or different SM
  - Run in order or out of order
- Should not assume anything on execution order of blocks
Block scheduling

- Blocks may
  - Run serially or in parallel
  - Run on the same or different SM
  - Run in order or out of order
- Should not assume anything on execution order of blocks
Outline

- GPU programming environments
- CUDA basics
  - Host side
  - Device side: threads, blocks, grids
- Expressing parallelism
  - Vector add example
- Managing communications
  - Parallel reduction example
Example: vector addition

- Addition example: only 1 thread  
  - Now let's run a parallel computation
- Start with multiple blocks, 1 thread/block  
  - Independent computations in each block
- No communication/synchronization needed
Host code: initialization

- A and B are now arrays: just change allocation size

```c
int main()
{
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);
    Initialize(h_A, h_B);

    // Allocate device memory
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **)&d_A, size);
    cudaMalloc((void **)&d_B, size);
    cudaMalloc((void **)&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_C, h_C, size, cudaMemcpyHostToDevice);

    ... }```
Host code: kernel and kernel launch

```c
__global__ void vectorAdd2(float *A, float *B, float *C) {
    int i = blockIdx.x;
    C[i] = A[i] + B[i];
}
```

- Launch n blocks of 1 thread each (for now)

```c
int blocks = numElements;
vectorAdd2<<<blocks, 1>>>(d_A, d_B, d_C);
```
\begin{verbatim}
__global__ void vectorAdd2(float *A, float *B, float *C) {
    int i = blockIdx.x;
    C[i] = A[i] + B[i];
}
\end{verbatim}

- Block number $i$ processes element $i$
- Grid of blocks may have up to 3 dimensions ($\text{blockIdx}.x$, $\text{blockIdx}.y$, $\text{blockIdx}.z$)
  - For programmer convenience: no effect on scheduling

Grid of blocks may have up to 3 dimensions
($\text{blockIdx}.x$, $\text{blockIdx}.y$, $\text{blockIdx}.z$)
Multiple blocks, multiple threads/block

Fixed number of threads / block: here 64

- **Host code**

  ```cpp
  int threads = 64;
  int blocks = (numElements + threads - 1) / threads; // Round up

  vectorAdd3<<<blocks, threads>>>(d_A, d_B, d_C, numElements);
  ```

- **Device code**

  ```cpp
  __global__ void vectorAdd3(const float *A, const float *B, float *C, int n)
  {
      int i = blockIdx.x * blockDim.x + threadIdx.x;

      if(i < n) {
          C[i] = A[i] + B[i];
      }
  }
  ```

  Not necessarily multiple of block size!

  Last block may have less work to do

  Thread block may also have up to 3 dimensions: threadIdx.{x,y,z}
Outline

- GPU programming environments
- CUDA basics
  - Host side
  - Device side: threads, blocks, grids
- Expressing parallelism
  - Vector add example
- Managing communications
  - Parallel reduction example
Barriers

- Threads can synchronize inside one block
- In C for CUDA:
  ```c
  __syncthreads();
  ```
- Needs to be called at the same place for all threads of the block

```c
if(tid < 5) {
  ...
} else {
  ...
}__syncthreads();
if(a[0] == 17) {
  __syncthreads();
} else {
  __syncthreads();
}__syncthreads();
if(tid < 5) {
  __syncthreads();
} else {
  __syncthreads();
}__syncthreads();
```

OK  OK  Wrong

Same condition for all threads in the block
Shared memory

- Fast, software-managed memory
  - Faster than global memory
- Valid only inside one block
  - Each block sees its own copy
- Used to exchange data between threads
- Concurrent writes: one thread wins, but we do not know which one
Thread communication: common pattern

- Each thread writes to its own location
  - No write conflict
- Barrier
  - Wait until all threads have written
- Read data from other threads
Example: parallel reduction

- Algorithm for 2-level multi-BSP model

![Diagram of parallel reduction]

- Level 1 reduction
- Level 2 reduction
- L1 barrier
- L2 Barrier
__global__ void reduce1(float *g_idata, float *g_odata, unsigned int n) 
{
    extern __shared__ float sdata[];  
    Dynamic shared memory allocation: will specify size later

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Load from global to shared mem
    sdata[tid] = (i < n) ? g_idata[i] : 0;
    __syncthreads();

    for(unsigned int s = 1; s < blockDim.x; s *= 2) {
        int index = 2 * s * tid;

        if(index < blockDim.x) {
            sdata[index] += sdata[index + s];
        }
        __syncthreads();
    }

    // Write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}

cuda/samples/6_Advanced/reduction
Reduction: host code

```c
int smemSize = threads * sizeof(float);
reduce1<<<blocks, threads, smemSize>>>(d_idata, d_odata, size);
```

Optional parameter:
Size of dynamic shared memory per block

- Level 2: run reduction kernel again, until we have 1 block left
- By the way, is our reduction operator associative?
A word on floating-point

- Parallel reduction requires the operator to be associative
- Is addition associative?
  - On reals: yes
  - On floating-point numbers: no
    With 4 decimal digits:
    \((1.234+123.4)-123.4=124.6-123.4=1.200\)

- Consequence: different result depending on thread count
Recap

- Memory management:
  Host code and memory / Device code and memory
- Writing GPU Kernels
- Dimensions of parallelism: grids, blocks, threads
- Memory spaces: global, local, shared memory

- Next time: code optimization techniques
References and further reading

- CUDA C Programming Guide