GPU programming: CUDA

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
- Re-using data
  - Matrix multiplication example
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
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 a = 1515, b = 159;  // Inputs, in host mem
    float c;
    // c = a + b;
    printf("c = %f\n", c);
}
```

vectorAdd example: cuda/samples/0_Simple/vectorAdd
int main()
{
    float a = 1515, b = 159;  // Inputs, in host mem
    // Allocate GPU memory
    size_t size = sizeof(float);
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **)&d_A, size);
    cudaMalloc((void **)&d_B, size);
    cudaMalloc((void **)&d_C, size);

    // Free GPU memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Allocate space for a and b in GPU memory
    // At the end, free memory

    Passing a pointer to the pointer to be overwritten
}
Step 2, 4: copy data to/from GPU memory

```c
int main()
{
  float a = 1515, b = 159;  // Inputs, CPU mem

  // Allocate GPU memory
  size_t size = sizeof(float);
  float *d_A, *d_B, *d_C;
  cudaMalloc((void **)&d_A, size);
  cudaMalloc((void **)&d_B, size);
  cudaMalloc((void **)&d_C, size);

  // Copy from CPU mem to GPU mem
  cudaMemcpy(d_A, &a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, &b, size, cudaMemcpyHostToDevice);

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

  // Free GPU memory
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}
```
Step 3: launch kernel

```c
__global__ void addOnGPU(float * a, float * b, float * c) {
    *c = *a + *b;
}

int main() {
    float a = 1515, b = 159;  // Inputs, CPU mem
    // Allocate GPU memory
    size_t size = sizeof(float);
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **)&d_A, size);
    cudaMalloc((void **)&d_B, size);
    cudaMalloc((void **)&d_C, size);
    // Copy from CPU mem to GPU mem
    cudaMemcpy(d_A, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, &b, size, cudaMemcpyHostToDevice);
    // Launch computation on GPU
    addOnGPU<<<1, 1>>>(d_A, d_B, d_C);

    float c;  // Result on CPU
    // Copy results back to CPU mem
    cudaMemcpy(&c, d_C, size, cudaMemcpyDeviceToHost);
    printf("c = %f\n", c);
    // Free GPU memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

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

What is inside the `<<<>>>`?
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
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 launch on a grid: `my_kernel<<<n, m>>>(...)`
- Two nested levels
  - n blocks / grid
  - m threads / block
Outer level: grid of blocks

- Blocks also named Concurrent Thread Arrays (CTAs)
- Virtually **unlimited** number of blocks
- **No communication** between blocks of the same grid
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 ∈ cores ∈ nodes...
  - Remember the memory wall, the speed of light
- Multi-BSP: BSP generalization with multiple nested levels

- Higher level: more expensive synchronization
Minor difference: BSP is based on message passing, CUDA on shared memory
SM resources are partitioned across 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
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
- Re-using data
  - Matrix multiplication 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;
    cudaMemcpy((void **)&d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy((void **)&d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMemcpy((void **)&d_C, h_C, size, cudaMemcpyHostToDevice);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    ...
}
```
__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)

    int blocks = numElements;
    vectorAdd2<<<blocks, 1>>>(d_A, d_B, d_C);
Device code

```c
__global__ void vectorAdd2(float *A, float *B, float *C)
{
    int i = blockIdx.x;  // Built-in CUDA variable: in device code only
    C[i] = A[i] + B[i];
}
```

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

Fixed number of threads / block: here 64

- Host code

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

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

- Device code

```c
__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];
    }
}
```

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
- Re-using data
  - Matrix multiplication 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();
```

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

```c
if(tid < 5) {
  __syncthreads();
} else {
  __syncthreads();
}
```

Same condition for all threads in the block.

OK  OK  Wrong
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 showing parallel reduction algorithm with processors P_0 to P_7 and variables a_0 to a_7. The diagram includes L1 barriers, L1 reduction, L2 Barrier, and Level 2 reduction.]
Reduction in CUDA: level 1

```c
__global__ void reduce1(float *g_idata, float *g_odata, unsigned int n) {
    extern __shared__ float sdata[];

    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];
}
```

Dynamic shared memory allocation: will specify size later
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
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
Arithmetic intensity

- Ratio of computation to memory pressure
  - High arithmetic intensity: compute bound algorithm
  - Low arithmetic intensity: memory bound algorithm

- Thursday lab work
  - NVIDIA GTX 470 needs 65 flops / word to be compute-bound

- How to reach enough arithmetic intensity?
  - Need to **reuse** values loaded from memory
Classic example: matrix multiplication

- Naive algorithm

```plaintext
for i = 0 to n-1
    for j = 0 to n-1
        for k = 0 to n-1
            C[i,j] += A[i,k] * B[k,j]
```

- Arithmetic intensity: 1:1   :(

![Diagram of matrix multiplication](image_url)
Reusing inputs

- Move loop on $k$ up

```
for k = 0 to n-1
  for i = 0 to n-1
    for j = 0 to n-1
      C[i,j] += A[i,k] * B[k,j]
```

- Enable data reuse on inputs
- But need to keep all matrix C in registers!
With tiling

- Block loops on i and j

  for i = 0 to n-1 step 16
    for j = 0 to n-1 step 16
      for k = 0 to n-1
        for i2 = i to i+15
          for j2 = j to j+15
            C[i2,j2] += A[i2,k] * B[k,j2]

- For one block: product between horizontal panel of A and vertical panel of B
With more tiling

- Block loop on k

```plaintext
for i = 0 to n-1 step 16
  for j = 0 to n-1 step 16
    for k = 0 to n-1 step 16
      for k2 = k to k+15
        for i2 = i to i+15
          for j2 = j to j+15
            C[i2,j2] += A[i2,k] * B[k,j2]
```

```
A

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>k</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

Constant size
Pre-loading data

for $i = 0$ to $n-1$ step 16
    for $j = 0$ to $n-1$ step 16
        $c = \{0\}$
        for $k = 0$ to $n-1$ step 16
            $a = A[i..i+15,k..k+15]$  
            $b = B[k..k+15,j..j+15]$  
            for $k2 = 0$ to $15$
                for $i2 = 0$ to $15$
                    for $j2 = 0$ to $15$
                        $c[i2,j2] += a[i2,k2] \times b[k2,j2]$  

$C[i..i+15,j..j+15] = c$

Load submatrices a and b
Multiply submatrices
$c = a \times b$
Store submatrix c
Breaking into two levels

- Run loops on i, j, i2, j2 in parallel

```plaintext
for // i = 0 to n-1 step 16
    for // j = 0 to n-1 step 16
        c = \{0\}
        for k = 0 to n-1 step 16
            a = A[i..i+15,k..k+15]
            b = B[k..k+15,j..j+15]
            for k2 = 0 to 15
                for // i2 = 0 to 15
                    for // j2 = 0 to 15
                        c[i2,j2] += a[i2,k2] * b[k2,j2]
            C[i..i+15,j..j+15] = c
```

- Let's focus on threads
Level 1: SIMD (PRAM-style) version

- Each processor has ID \((x,y)\)
  - Loops on \(i_2, j_2\) are implicit

\[
c[x,y] = 0 \\
\text{for } k = 0 \text{ to } n-1 \text{ step 16} \\
a[x,y] = A[i+x,k+y] \\
b[x,y] = B[k+x,j+y] \\
\text{for } k_2 = 0 \text{ to } 15 \\
c[x,y] += a[x,k_2] \times b[k_2,y] \\
C[i+x,j+y] = c[x,y]
\]

Load submatrices \(a\) and \(b\)
Multiply submatrices \(c = a \times b\)
Store submatrix \(c\)

Private writes: no conflict
Read from other processors

- How to translate to SPMD (BSP-style)?
SPMD version

- Place synchronization barriers

\[
\begin{align*}
c[x,y] &= 0 \\
\text{for } k = 0 \text{ to } n-1 \text{ step 16} & \\
\quad a[x,y] &= A[i+x,k+y] \\
\quad b[x,y] &= B[k+x,j+y] \\
\text{Barrier} & \\
\quad \text{for } k2 = 0 \text{ to } 15 & \\
\quad c[x,y] &= a[x,k2]*b[k2,y] \\
\text{Barrier} & \\
\quad C[i+x,j+y] &= c[x,y]
\end{align*}
\]

- Why do we need the second barrier?
Data allocation

- 3 memory spaces: **Global, Shared, Local**

- Where should we put: A, B, C, a, b, c?

```
c[x,y] = 0
for k = 0 to n-1 step 16
    a[x,y] = A[i+x,k+y]
    b[x,y] = B[k+x,j+y]
    Barrier
    for k2 = 0 to 15
        c[x,y] += a[x,k2]*b[k2,y]
    Barrier
C[i+x,j+y] = c[x,y]
```
Data allocation

- Memory spaces: **Global**, **Shared**, **Local**
  - As local as possible

```plaintext
\[ c = 0 \]
for \( k = 0 \) to \( n-1 \) step 16
  \[ a[x,y] = A[i+x,k+y] \]
  \[ b[x,y] = B[k+x,j+y] \]
  Barrier
  for \( k2 = 0 \) to \( 15 \)
    \[ c += a[x,k2] \times b[k2,y] \]
  Barrier
\[ C[i+x,j+y] = c \]
```

- **Local**: private to each thread (indices are implicit)
- **Global**: shared between blocks / inputs and outputs
- **Shared**: shared between threads, private to block
CUDA version

- Straightforward translation

```c
float Csub = 0;

for(int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];

    __syncthreads();
    for(int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub += As[ty][k] * Bs[k][tx];
    }
    __syncthreads();
}

int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```

matrixMul example: cuda/samples/0_Simple/matrixMul

- Precomputed base addresses
- Declare shared memory
- Linearized arrays
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
- Extra features
Multiple grid/block dimensions

- Grid and block size are of type `dim3`
  - Support up to 3 dimensions
    ```
    dim3 dimBlock(tx, ty, tz);
    dim3 dimGrid(bx, by, bz);
    my_kernel<<<dimGrid, dimBlock>>>(arguments);
    ```
  - Implicit cast from int to `dim3`
    y and z sizes are 1 by default
- On device side, `threadIdx`, `blockDim`, `blockIdx`, `gridDim` are also of type `dim3`
  - Access members with `.x`, `.y`, `.z`
- Some early GPUs (CC 1.x) only support 2-D grids
Local memory

- Registers are fast but
  - Limited in size
  - Not addressable

- Local memory used for
  - Local variables that do not fit in registers (*register spilling*)
  - Local arrays accessed with indirection
    
    ```
    int a[17];
    b = a[i];
    ```

- **Warning**: local is a misnomer!
  - Physically, local memory usually goes off-chip
Device functions

- Kernel can call functions
- Need to be marked for GPU compilation

```
__device__ int foo(int i) {
}
```

- A function can be compiled for both host and device

```
__host__ __device__ int bar(int i) {
}
```

- Device functions can call device functions
  - Older GPUs do not support recursion
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()
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: advanced features and optimization techniques
References and further reading

- CUDA C Programming Guide