Introduction to CUDA

Oliver Meister

November 7th 2012
References

- D. Kirk, W. Hwu: *Programming Massively Parallel Processors*, Morgan Kaufmann, 2010
Motivation

Currently there are two opportunities of parallelizing programs multi-cores

- distribute the work on few strong multi-purpose processors (CPUs)
- regular supercomputers, clusters
- OpenMP, MPI

many-cores

- distribute the work on a lot of single purpose processors
- Intel MIC, BlueGene, GPUs
CPU
- general purpose
- large amount of transistors for non-computational tasks
- allows out of order execution
- pipelining
- optimized for sequential tasks

GPU
- many processors dedicated to computations
- less support for branching
- well aligned data streamed through processor – data parallelism
GPU Computing – Origins

Fixed-function graphics pipelines:
- ’80s/’90s: hardware configurable, but not programmable
- implementation of graphics APIs (OpenGL, DirectX, etc.)
- vertex shading/transform/lighting, raster operations, textures, etc.

Programmable Real-Time Graphics:
- shader programmability, floating-point pixel/shader/vertex processing
- resp. API extensions in DirectX, OpenGL
- programmable pipeline stages; hardware evolves towards massively parallel architectures
"GPGPU":
- general purpose computing on GPUs
- implement non-graphical algorithms/computations via shader functions
- driven by performance advantage of GPUs

GPU Computing:
- hardware-side: general trend towards "many-core"; GPUs evolve towards massively parallel, wider-purpose architectures
- software-side: programming models for GPU computing: CUDA, OpenCL, ...
## Different Programming Models for GPU

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>• GPU - only</td>
<td>• standard formed by consortium (Khronos Group)</td>
</tr>
<tr>
<td>• standard formed by vendor (nVidia)</td>
<td>• platform independent (also for ATI and CPU’s)</td>
</tr>
<tr>
<td>• adopts new architectures fast</td>
<td>• slower development</td>
</tr>
</tbody>
</table>

---

We will use CUDA

- interface is easier to learn
- paradigms of GPU programming better understandable
GPU Architectures – NVIDIA Kepler

(source: NVIDIA – Kepler Whitepaper)
The third generation SM introduces several architectural innovations that make it not only the most powerful SM yet built, but also the most programmable and efficient.

- 512 High Performance CUDA cores
- Each SM features 32 CUDA processors—a fourfold increase over prior SM designs. Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU).
- Prior GPUs used IEEE 754-1985 floating point arithmetic. The Fermi architecture implements the new IEEE 754-2008 floating-point standard, providing the fused multiply-add (FMA) instruction for both single and double precision arithmetic. FMA improves over a multiply-add (MAD) instruction by doing the multiplication and addition with a single final rounding step, with no loss of precision in the addition. FMA is more accurate than performing the operations separately.
- GT200 implemented double precision FMA.
- In GT200, the integer ALU was limited to 24-bit precision for multiply operations; as a result, multi-instruction emulation sequences were required for integer arithmetic. In Fermi, the newly designed integer ALU supports full 32-bit precision for all instructions, consistent with standard programming language requirements. The integer ALU is also optimized to efficiently support 64-bit and extended precision operations. Various instructions are supported, including Boolean, shift, move, compare, convert, bit-field extract, bit-reverse insert, and population count.

- 16 Load/Store Units
- Each SM has 16 load/store units, allowing source and destination addresses to be calculated for sixteen threads per clock. Supporting units load and store the data at each address to cache or DRAM.

(source: NVIDIA – Fermi/Kepler Whitepapers)
GPU Architectures – NVIDIA Kepler (3)

(source: NVIDIA – Kepler Whitepaper)
CUDA – Architecture Model

Host & Device:
- host = regular CPU, main memory
- device(s) = GPU/coprocessor(s) with separate memory

Hardware characteristics:
- massively parallel (hundreds of cores)
- lightweight threads, hardware-supported; typically multiple threads assigned to a single core
- massive parallelism hides memory latency; focus on data parallelism
Warps

- 32 (16) threads executed in parallel
- only one instruction possible per cycle and warp
- if a branch occurs only one part of the warp is executed

Host Memory

- slowly accessible
- reduce access to host memory
CUDA – Programming Model

CUDA as extension of C:

- host code (program control) and device code (GPU) combined in a single C program
- device code consists of massively parallel *kernels* that are off-loaded to the GPU
- language extension for defining and calling kernels
- API function to allocate device/host memory, synchronise threads, etc.
- SIMD/SPMD (single instruction/program, multiple data)
Example: Matrix Multiplication

General Approach: PRAM program

\[
\text{for } i \text{ from } 1 \text{ to } n \text{ do (in parallel?) }
\]
\[
\quad \text{for } k \text{ from } 1 \text{ to } n \text{ do (in parallel?) }
\]
\[
\quad \quad \text{for } j \text{ from } 1 \text{ to } n \text{ do (in parallel?) }
\]
\[
\quad \quad \quad C[i,k] += A[i,j] \times B[j,k]
\]
Example: Matrix Multiplication

General Approach: PRAM program

\[
\text{for } i \text{ from } 1 \text{ to } n \text{ do in parallel} \\
\quad \text{for } k \text{ from } 1 \text{ to } n \text{ do in parallel} \\
\quad \quad \text{for } j \text{ from } 1 \text{ to } n \text{ do} \\
\quad \quad \quad C[i,k] += A[i,j] \times B[j,k]
\]

- PRAM: executed on \( n^2 \) processors
- CUDA: \( n^2 \) CUDA threads; each thread executes one j-loop (i.e., computes one element \( C[i,k] \))
- part 1: memory transfer (host \( \rightarrow \) device and device \( \rightarrow \) host)
- part 2: launch/execution of kernel code for j-loop
CUDA Memory transfer

Memory instructions in CUDA:

- Memory allocation:
  \texttt{cudaMalloc(void** \textit{ppd}, int \textit{size});}

- Memory deallocation:
  \texttt{cudaFree(void* \textit{pd});}

- Copy from host to device:
  \texttt{cudaMemcpy(pd, p, size, cudaMemcpyHostToDevice);}

- Copy from device to host:
  \texttt{cudaMemcpy(p, pd, size, cudaMemcpyDeviceToHost);}
Matrix Multiplication – Memory Transfer

```c
__host__ void matrixMult(float *A, float *B,
                         float *C, int n);
```

Task:

- Write the memory transfer code for a matrix multiplication $C = A \cdot B$, i.e. allocate device data, copy data between host and device and deallocate device data. All matrices have size $n \times n$. 
Matrix Multiplication – Memory Transfer

```c
__host__ void matrixMult(float *A, float *B,
    float *C, int n) {
    int size = n*n*sizeof(float);
    float* Ad; float* Bd; float* Cd;
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Cd, C, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Cd, C, size, cudaMemcpyHostToDevice);
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
    cudaFree(Ad); cudaFree(Bd); cudaFree(Cd);
}
```
Matrix Multiplication – CUDA Kernel

```c
__global__ void matrixMultKernel(float* Ad, float* Bd, float* Cd, int n) {

    int i = threadIdx.x;
    int k = threadIdx.y;
    float Celem = 0;
    for(int j=0; j<n; j++) {
        float Aelem = Ad[i*n+j];
        float Belem = Bd[j*n+k];
        Celem += Aelem*Belem;
    }
    Cd[i*n+k] += Celem;
}
```
Kernel Invocation: Grids and Blocks

/* ... */
dim3 dimBlock(n,n);
dim3 dimGrid(1,1);
matrixMultKernel<<dimGrid,dimBlock>>>(Ad,Bd,Cd,n);
/* ... */

• threads are combined to 3D blocks:
  → threadIdx.x, threadIdx.y, threadIdx.z

• blocks are combined to 2D grids:
  → blockIdx.x, blockIdx.y
Grids and Blocks in CUDA

Blocks:

- threads can be organised as 1D, e.g. (128,1,1), 2D, e.g. (16,16,1), or 3D, e.g. (4,8,16) blocks
- limited to 1024 (Fermi, Kepler), 512 (GT2xx) threads per block
- threads in one block are always executed in parallel
- and can use separate, shared memory

Grids:

- \texttt{dim3}, but 2D layout (3rd component ignored)
- up to $2^{32} \times 2^{32}$ (Kepler) / $2^{16} \times 2^{16}$ ($\leq$ Fermi) blocks per grid
- blocks in a grid may be executed in parallel
  (but, in practice, will be scheduled to available cores)
Matrix Multiplication – with Grid

```c
__global__ void matrixMultKernel(float* Ad, float* Bd, float* Cd, int n);
```

Task:
- Extend the kernel from a single block to a grid with multiple blocks. Each block should have the size $TILE_SIZE \times TILE_SIZE$.
- Change the definition of grid and block dimensions accordingly
Matrix Multiplication – with Grid

```c
__global__ void matrixMultKernel(float* Ad, float* Bd, float* Cd, int n) {
    int i = blockIdx.x * TILE_SIZE + threadIdx.x;
    int k = blockIdx.y * TILE_SIZE + threadIdx.y;
    float Celem = 0;
    for(int j=0; j<n; j++) {
        float Aelem = Ad[i*n+j];
        float Belem = Bd[j*n+k];
        Celem += Aelem*Belem;
    }
    Cd[i*n+k] += Celem;
}
```
Matrix Multiplication – with Grid (2)

/* ... */
dim3 dimBlock(TILE_SIZE,TILE_SIZE);
dim3 dimGrid(n/TILE_SIZE,n/TILE_SIZE);
matrixMultKernel<<<dimGrid,dimBlock>>>(Ad,Bd,Cd,n);
/* ... */

• What is the optimal tile size?

• Too small → large overhead, low performance
• Too large → block size limit

\[ \text{TILE_SIZE}^2 \leq \text{max. threads per block} \]
⇒ choose \( \text{TILE_SIZE} = 32 \) (Fermi, Kepler), 16 (for GT2xx)

• in practice: padding of matrix to match tile size (\( n = k \cdot 32 \))
Matrix Multiplication – with Grid (2)

/* ... */
dim3 dimBlock(TILE_SIZE,TILE_SIZE);
dim3 dimGrid(n/TILE_SIZE,n/TILE_SIZE);
matrixMultKernel<<<dimGrid,dimBlock>>>(Ad,Bd,Cd,n);
/* ... */

- What is the optimal tile size?
- Too small → large overhead, low performance
- Too large → block size limit
- \( \text{TILE\_SIZE}^2 \leq \text{max. threads per block} \)
  \( \Rightarrow \) choose \( \text{TILE\_SIZE} = 32 \) (Fermi, Kepler), 16 (for GT2xx)
- in practice: padding of matrix to match tile size \( n = k \cdot 32 \)
Matrix Multiplication – with Grid (2)

/* ... */
dim3 dimBlock(TILE_SIZE,TILE_SIZE);
dim3 dimGrid(n/TILE_SIZE,n/TILE_SIZE);
matrixMultKernel<<dimGrid,dimBlock>>>(Ad,Bd,Cd,n);
/* ... */

• What is the optimal tile size?
• Too small $\rightarrow$ large overhead, low performance
• Too large $\rightarrow$ block size limit
• $TILE\_SIZE^2 \leq$ max. threads per block
  $\Rightarrow$ choose $TILE\_SIZE = 32$ (Fermi, Kepler), 16 (for GT2xx)
• in practice: padding of matrix to match tile size ($n = k \cdot 32$)

Let’s compare CPU and GPU implementation.
Matrix Multiplication – with Grid (2)

/* ... */
dim3 dimBlock(TILE_SIZE,TILE_SIZE);
dim3 dimGrid(n/TILE_SIZE,n/TILE_SIZE);
matrixMultKernel<<<dimGrid,dimBlock>>>(Ad,Bd,Cd,n);
/* ... */

- What is the optimal tile size?
- Too small $\rightarrow$ large overhead, low performance
- Too large $\rightarrow$ block size limit
- $TILE\_SIZE^2 \leq$ max. threads per block
  $\Rightarrow$ choose $TILE\_SIZE = 32$ (Fermi, Kepler), 16 (for GT2xx)
- in practice: padding of matrix to match tile size ($n = k \cdot 32$)

Let’s compare CPU and GPU implementation.
$\Rightarrow$ Okay, that sucks.
CUDA Memory

Types of device memory in CUDA:

- per thread: registers and local memory (locally declared variables and arrays (local memory), → lifetime: kernel execution)
- per block: shared memory (keyword __shared__, lifetime: kernel execution)
- per grid: global memory and constant memory (keywords __device__, __constant__; lifetime: entire application)
- vs.: CPU main memory (host memory)
Matrix Multiplication – Performance Estimate

Multiplication kernel:

```c
for(int j=0; j<n; j++) {
    float Aelem = Ad[i*n+j];
    float Belem = Bd[j*n+k];
    Celem += Aelem*Belem;
}
```

- NVidia NVS 5200M stats: memory bandwidth: 14.4 GB/s, floating point performance: 240 GFlop/s
- two floating-point operations (multiply and add) per two floating-point variables (each 4 byte)
- thus: max. of 3.6 giga float variables can be transferred from global memory per second
- limits performance to < 4 GFlop/s
Matrix Multiplication with Tiling

• observation: simple matrix multiplication kernel is slow (far below peak performance)
• anticipated reason: only access to slow global memory; performance limited by memory bandwidth between global memory and CUDA cores

Remedy: Tiling

• switch to tile-oriented implementation (matrix multiplication on sub-blocks)
• copy matrix tiles into shared memory
• let all threads of a block work together on shared tile
• accumulate result tile back on matrix in global memory
A Note on Synchronisation

Barrier-synchronisation in CUDA:

__syncthreads();

- barrier for all threads within a block
- usual rules: all threads need to execute (or not) the same(!) call to __syncthreads()
- threads of the same block scheduled to the same hardware unit
- in contrast: no synchronisation features for threads in a grid → reason: *transparent scheduling* of entire blocks
Matrix Multiplication – with Tiles

__global__ void matrixMultKernel(float* Ad, float* Bd, float* Cd, int n) {

    __shared__ float Ads[TILE_SIZE][TILE_SIZE];
    __shared__ float Bds[TILE_SIZE][TILE_SIZE];

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int i = blockIdx.x * TILE_SIZE + tx;
    int k = blockIdx.y * TILE_SIZE + ty;

    /* (cont.) */

Task:

• Copy matrix tiles of A and B from global to shared memory.
• Execute matrix multiplication on shared tiles and write the result to matrix C in global memory.
• Synchronize the thread block where it is necessary.