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
- regular supercomputers, clusters
- OpenMP, MPI

**many-cores**
- distribute the work on a lot of single purpose processors
- Larrabee (discontinued), 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

![Diagram comparing CPU and GPU architecture](image-url)
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
GPU Computing – Origins (2)

“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

for i from 1 to n do (in parallel?)
  for k from 1 to n do (in parallel?)
    for j from 1 to n do (in parallel?)
      C[i,k] += A[i,j]*B[j,k]
Example: Matrix Multiplication

General Approach: PRAM program

```plaintext
for i from 1 to n do in parallel
    for k from 1 to n do in parallel
        for j from 1 to n do
            C[i,k] += A[i,j]*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
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;
    cudaMalloc((void**)&Ad, size);
    cudaMalloc((void**)&Bd, size);
    cudaMalloc((void**)&Cd, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Cd, C, size, cudaMemcpyHostToDevice);
    /* ... perform multiplication on device ... */
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
    cudaFree(Ad); cudaFree(Bd); cudaFree(Cd);
}
```
Matrix Multiplication – CUDA Kernel

```
__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

```c
/* ... */
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:**

- `dim3`, but 2D layout (3rd component ignored)
- up to $2^{32} \times 2^{32}$ (Kepler) / $2^{16} \times 2^{16}$ (≤ 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) {
    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)

```cpp
/* ... */
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 (\( n = k \cdot 32 \))

Let's compare CPU and GPU implementation.
⇒ Okay, that sucks.
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
- TILE_SIZE^2 ≤ max. threads per block
  ⇒ 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 → large overhead, low performance
• Too large → block size limit
• TILE_SIZE^2 ≤ max. threads per block
  ⇒ choose TILE_SIZE = 32 (Fermi, Kepler), 16 (for GT2xx)
• in practice: padding of matrix to match tile size (n = k · 32)

Let’s compare CPU and GPU implementation.
⇒ Okay, that sucks.
Assignments

1. Implement a simple matrix–matrix multiplication using CUDA for small matrices (up to $n = 16$). Write the corresponding host and device code (see page 16 and 17). Compare the time with the time required by a regular CPU.

2. Extend the code using different block sizes (see page 20) for the computation. Again do a measurement of the execution time and compare it to the previous results. Then increase the system size and do further comparisons to the execution times on the CPU.
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;
}
```

- memory bandwidth: 159 GB/s (for GeForce GTX 285)
- two floating-point operations (multiply and add) per two floating-point variables (each 4 byte)
- thus: max. of 40 giga float variable can be transferred from global memory per second
- limits performance to < 40GFlop/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
Matrix Multiplication – with Tiles

```c
__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;
for(int m=0; m < n/TILE_SIZE; m++) {
    Ads[tx][ty] = Ad[ i*n + m*TILE_SIZE+ty];
    Bds[tx][ty] = Bd[ (m*TILE_SIZE+tx)*n + k];
}
/* perform matrix multiplication on shared tiles */
```
Matrix Multiplication – with Tiles

/* (cont.) */
for(int m=0; m < n/TILE_SIZE; m++) {
    Ads[tx][ty] = Ad[ i*n + m*TILE_SIZE+ty];
    Bds[tx][ty] = Bd[ (m*TILE_SIZE+tx)*n + k];
    __syncthreads();
    /* perform matrix multiplication on shared tiles */
    for(int j=0; j<TILE_SIZE; j++)
        Celem += Ads[tx][j]*Bds[j][ty];
        __syncthreads();
    }
    Cd[i*n+k] += Celem;
}
Matrix Multiplication – with Tiles

/* (cont.) */

for(int m=0; m < n/TILE_SIZE; m++) {
    Ads[tx][ty] = Ad[ i*n + m*TILE_SIZE+ty];
    Bds[tx][ty] = Bd[ (m*TILE_SIZE+tx)*n + k];
    __syncthreads();
    /* perform matrix multiplication on shared tiles */
    for(int j=0; j<TILE_SIZE; j++)
        Celem += Ads[tx][j]*Bds[j][ty];
    __syncthreads();
}
Cd[i*n+k] += Celem;

Let’s see what happens.
Matrix Multiplication – with Tiles

/* (cont.) */

for(int m=0; m < n/TILE_SIZE; m++) {
    Ads[tx][ty] = Ad[ i*n + m*TILE_SIZE+ty];
    Bds[tx][ty] = Bd[ (m*TILE_SIZE+tx)*n + k];
    __syncthreads();
    /* perform matrix multiplication on shared tiles */
    for(int j=0; j<TILE_SIZE; j++)
        Celem += Ads[tx][j]*Bds[j][ty];
    __syncthreads();
}
Cd[i*n+k] += Celem;

Let’s see what happens. ⇒ Still bad.
Updated Performance Estimate

• at start, each thread loads one matrix element from global memory
• shared memory → no further loads in TILE_SIZE m-iterations
• we reduce the memory transfer from global memory to $1/TILE\_SIZE$
• for $TILE\_SIZE = 16$: new performance limit at $640\, GFlop/s$
  → we’ve eliminated a major bottleneck, but apparently hit another ...
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
Assignments

3. Extend the previous program by the tiling algorithm and compare its performance with the previous results.
CUDA requirements

- CUDA-capable Nvidia GPU:
- Up-to-date graphics drivers (current version: 304.64)
- CUDA Toolkit (current version: 5.0)
- (Windows) Microsoft Windows XP, Vista, 7, or 8 or Windows Server 2003 or 2008
- (Windows) Microsoft Visual Studio 2008 or 2010, or a corresponding version of Microsoft Visual C++ Express
- (Linux) Up-to-date Linux version with gcc
CUDA installation steps

For details, refer to the Getting Started Guides for Windows/Linux/Mac on https://developer.nvidia.com/cuda-downloads

- Download CUDA toolkit from here:
- Run installer, install drivers if necessary (Linux: new drivers require X server restart), samples may be skipped
- (Linux) PATH needs to include /usr/local/cuda/bin
- (Linux) LD_LIBRARY_PATH needs to contain /usr/local/cuda/lib and /usr/local/cuda/lib64 for x64 systems.
Compiling CUDA-code

- Open a (Linux) terminal / (Windows) Visual Studio command prompt, and type `nvcc <source files.cu>`
- That's it, run your code :)

Oliver Meister: Introduction to CUDA
Tutorial Parallel Programming and High Performance Computing, November 7th 2012
What if no CUDA capable device is available?

- CUDA emulator, works on CPU, AMD and Nvidia chips
- Not fully compatible with CUDA though
- Ubuntu 10.10 / 11.04: Debian packages available
- Otherwise: compile from source
Remote compilation

What if no CUDA capable device is available? (2)

- Remote login to one of these two machines:
  atsccs30.informatik.tu-muenchen.de or
  atsccs59.informatik.tu-muenchen.de

- Requires student account: Send me your full name, account name (TUM, in.tum, ...), email address

- You’ll get an email with instructions for the first login

- (Linux) Open an ssh connection via
  ssh -X atsccs30.informatik.tu-muenchen.de, and
download the exercise code (i.e. open firefox from the terminal).

- No further setup should be necessary to compile the code.

- If anything doesn’t work, contact me.