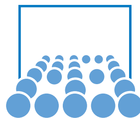
An Introduction to CUDA

Daniel Butnaru, Christoph Kowitz

November 7th 2011



References

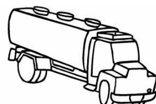
- D. Kirk, W. Hwu:
  Programming Massively Parallel Processors, Morgan Kaufmann, 2010

Motivation

Currently there are two opportunities of parallelizing programs

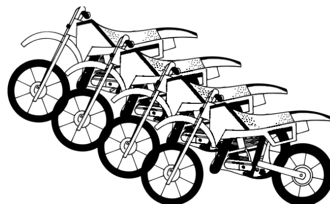
multi-cores

- 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
- Larabee, BlueGene, GPU's

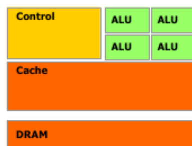


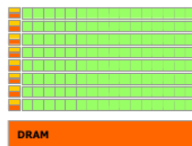
CPU

- 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:

- 80ies/90ies: 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

CUDA

- GPU - only
- standard formed by vendor (nVidia)
- adopts new architectures fast

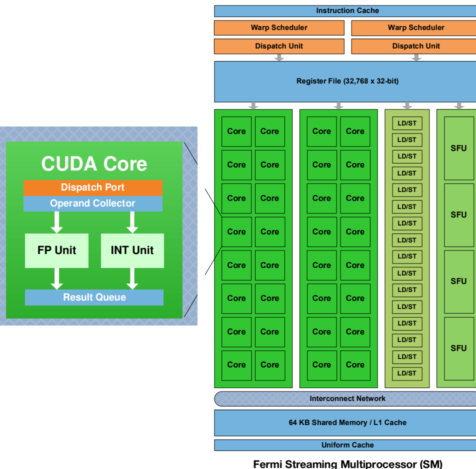
OpenCL

- standard formed by consortium (Khronos Group)
- platform independent (also for ATI and CPU's)
- slower development

We will use CUDA

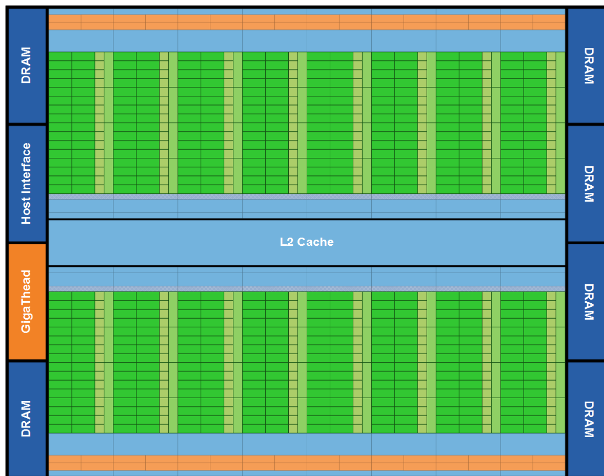
- interface is easier to learn
- paradigms of GPU programming better understandable

GPU Architectures – NVIDIA Fermi



(source: NVIDIA – Fermi Whitepaper)

GPU Architectures – NVIDIA Fermi (2)



(source: NVIDIA – Fermi Whitepaper)

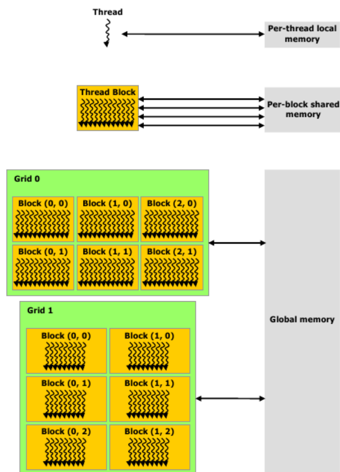
CUDA – Architecture Model

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
            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→device and device→host)
- part 2: launch/execution of kernel code for j-loop

Matrix Multiplication – Memory Transfer

```
__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

```
__host__ void matrixMult(float *A, float *B, float *C, int n)
{
    /* ... */
    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 512 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 65536 $\times$ 65536 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

```
__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)

```
__host__ void matrixMult(float *A, float *B, float *C, int n)
{
   /* ... */
   dim3 dimBlock(TILE_SIZE,TILE_SIZE);
   dim3 dimGrid(n/TILE_SIZE,n/TILE_SIZE);
   matrixMultKernel<<<dimGrid,dimBlock>>>(Ad,Bd,Cd,n);
   /* ... */
}
```

- choose TILE_SIZE = 16
  (“square” blocks and number of threads < 512)
- in practice: requires padding of matrix to match size (multiple of 16)
- works for large matrices – how about performance?

Assignments

1. Implement a simple matrix–vector multiplication using CUDA for small matrices (up to n=16). Write the corresponding host and device code (see page 14 and 15). Compare the time with the time required by a regular CPU.
2. Extend the code using different blocks (see page 18) for the computation. Again do a measurement of the execution time and compare it to the previous results. Then increase the systems size and do further comparisons to the executions time on the CPU.

CUDA Memory

Types of **device** memory in CUDA:

- per thread: **registers** and **local memory**
  (locally declared variables and arrays (local memory),
  $\rightarrow$ 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:

```
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 $< 40 GFlop/s$

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

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;
    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;
}
```

Updated Performance Estimate

- at start, each thread loads one matrix element from global memory
- shared memory $\rightarrow$ 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 $640 GFlop/s$

$\rightarrow$ we've eliminated a major bottleneck, but apparently hit another ...

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

Assignments

- Extend the previous program by the tiling algorithm and compare its performance with the previous results.