GPU Hardware and Parallelism

Jiří Filipovič

Fall 2013

Alternatives to CUDA

CUDA is (and probably will be) only for nVidia GPU.

Alternatives to CUDA

CUDA is (and probably will be) only for nVidia GPU.
OpenCL

- a standard for various types of accelerators (independent on HW vendor and OS)
- strongly inspired by CUDA, very easy transition

Alternatives to CUDA

CUDA is (and probably will be) only for nVidia GPU.

OpenCL

- a standard for various types of accelerators (independent on HW vendor and OS)
- strongly inspired by CUDA, very easy transition

DirectX compute

- Various GPU vendors, one OS

Alternatives to CUDA

CUDA is (and probably will be) only for nVidia GPU.

OpenCL

- a standard for various types of accelerators (independent on HW vendor and OS)
- strongly inspired by CUDA, very easy transition

DirectX compute

- Various GPU vendors, one OS

Brook(+)

- multi-platform, only for AMD/ATI
- only for streams

Why to learn CUDA?

Why CUDA and not OpenCL?

- published results still show higher speed
- better stability of the environment
- biggest number of applications
- biggest number of libraries
- biggest number of publications
- easier to learn
- similarity to OpenCL allows easy transition
- PGI x86 CUDA compiler

Differences among CUDA GPUs

New generations bring higher performance and new computing capabilities.

- *compute capability* describes richness of GPU instruction set and amount of resources such as registers, number of concurrently running threads, etc.
- the performance grows with the ability to put more than one core on a GPU

Cards in on generation also differ in performance substantially

- to produce more affordable cards
- due to changes introduces later in the manufacturing process
- to minimize power consumption of mobile GPUs

GPUs Available Today

Currently available GPUs

- compute capability 1.0 - 2.1
  - we will learn the differences later
- 1–30 multiprocessors (19.2 - 1 345.0 GFLOPs)
- frequency of 800 MHz–1.836 GHz
- width and speed of data bus sběrnice (64–512 bit, 6.4–177 GB/s)

Available products

GeForce graphics cards

- mainstream solution for gaming
- cheap, wildely used, broad range of performance
- disadvantage – limited memory (up to 1.5 GB on GPU)

Professional Quadro graphics cards

- the same as GeForce from CUDA perspective
- up to 4 GB of memory on GPU
- several times more expensive

Tesla

- a solution specially designed for CUDA computing
- one GPU per generation (basic variant), always large memory
- available as a PCIe card or standalone multi-GPU machines
- expensive, interesting for computing centers and personal supercomputers

GPU Parallelism

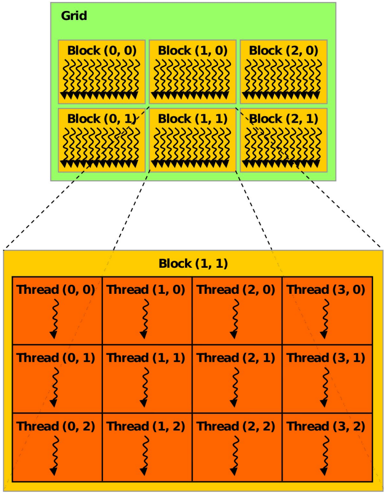
Parallel algorithms need to be designed w.r.t. the parallelism available in the HW

- GPU: array of SIMT multiprocessors working using shared memory

Decomposition for GPU

- coarse-grained decomposition of the problem into the parts that don't need intensive communication
- fine-grained decomposition similar to vectorization (but SIMT is more flexible)

Task Hierarchy



SIMT

A multiprocessor has one unit executing an instruction

- all 8 SPs have to execute the same instruction
- new instruction is executed every 4 cycles
- 32 threads (so called *warp*) need to execute the same instruction

SIMT

A multiprocessor has one unit executing an instruction

- all 8 SPs have to execute the same instruction
- new instruction is executed every 4 cycles
- 32 threads (so called *warp*) need to execute the same instruction

How about code branching?

- if different parts of a warp perform different instructions, they are serialized
- decreases performance—should be avoided

SIMT

A multiprocessor has one unit executing an instruction

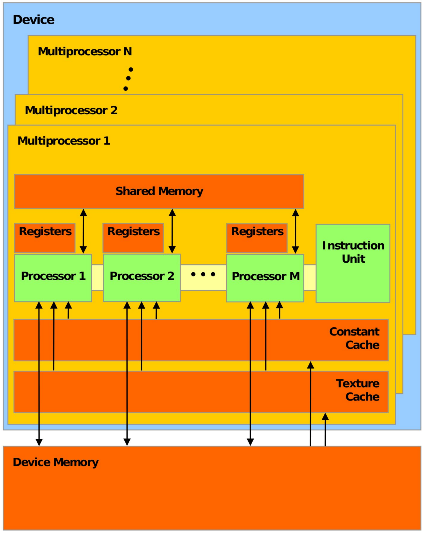
- all 8 SPs have to execute the same instruction
- new instruction is executed every 4 cycles
- 32 threads (so called *warp*) need to execute the same instruction

How about code branching?

- if different parts of a warp perform different instructions, they are serialized
- decreases performance—should be avoided

The multiprocessor is thus MIMD (Multiple-Instruction Multiple-Thread) from programmer's perspective and SIMT (Single-Instruction Multiple-Thread) from performance perspective.

GPU Architecture



Thread Properties

GPU threads are very lightweight compared to CPU threads.

- their run time can be very shorts (even tens of instructions)
- there may be (should be) many of them
- they should not use large amount of resources

Threads are aggregated into blocks

- blocks are run on individual multiprocessors
- having sufficient number of blocks is substantial to achieve good scalability

Number of threads and thread blocks per multiprocesor is limited.

Memory Latency Masking

Memory has latency

- global memory has high latency (hundreds of cycles)
- registers and shared memory have read-after-write latency

Memory Latency Masking

Memory has latency

- global memory has high latency (hundreds of cycles)
- registers and shared memory have read-after-write latency

Memory latency hiding is different from CPU

- no instructions are executed out of order
- most memory types have no cache

Memory Latency Masking

Memory has latency

- global memory has high latency (hundreds of cycles)
- registers and shared memory have read-after-write latency

Memory latency hiding is different from CPU

- no instructions are executed out of order
- most memory types have no cache

When a warp waits for data from memory, another warp may be executed

- allows memory latency hiding
- requires execution of *an order of magnitude more* threads compared to number of GPU cores
- thread execution scheduling and switching is implemented directly in HW without overhead

Memory Latency Masking

Memory has latency

- global memory has high latency (hundreds of cycles)
- registers and shared memory have read-after-write latency

Memory latency hiding is different from CPU

- no instructions are executed out of order
- most memory types have no cache

When a warp waits for data from memory, another warp may be executed

- allows memory latency hiding
- requires execution of *an order of magnitude more* threads compared to number of GPU cores
- thread execution scheduling and switching is implemented directly in HW without overhead

Works similarly for synchronization.

Thread-Local Memory

Registers

- fastest memory, directly usable in instructions
- local variables in a kernel and variables for intermediate results go automatically into the registers
  - if there is sufficient number of registers
  - if the compiler can determine static array indexing
- thread (warp) scoped

Thread-Local Memory

Registers

- fastest memory, directly usable in instructions
- local variables in a kernel and variables for intermediate results go automatically into the registers
    - if there is sufficient number of registers
    - if the compiler can determine static array indexing
- thread (warp) scoped

Local memory

- data that doesn't fit into the registers go into the local memory
- local memory is stored in DRAM $\Longrightarrow$ slow, high latency
- thread (warp) scoped

Block-Local Memory

Shared memory

- as fast as registers for c. c. 1.x
  - if memory bank conflicts are avoided
  - instructions can use only one operand in shared memory (otherwise explicit load/store is needed)
- declared using *__shared__* in C for CUDA
- a variable in shared memory can have dynamic size (determined at startup), if declared as *extern* withou size specification
- block scoped

Shared Memory

Static shared memory declaration

```
__shared__ float myArray[128];
```

Dynamic allocation

```
extern __shared__ char myArray[];
float *array1 = (float*)myArray;
int *array2 = (int*)&array1[128];
short *array3 = (short*)&array2[256];
```

It creates an array *array1* of *float* type with size 128, *array2* of *int* type sized 256, and *array3* of floating size. Total size has to be specified at kernel startup.

```
myKernel<<<grid, block, n>>>();
```

GPU Local Memory

Global memory

- an order of magnitude lower bandwidth compared to shared memory
- latency in order of hundreds for GPU cycles
- addressing needs to be aligned to get optimum performance
- application-scoped
- L1 cache (128 bytes/row) and L2 cache (32 bytes/row) in Fermi architecture

May be dynamically allocated using *cudaMalloc* or statically allocated using __*device*__ declaration.

GPU Local Memory

Constant memory

- read-only
- cached
- cache hit is as fast as registry (under certain constraints), cache miss is as fast as global memory
- limited size (64 kB for currently available GPUs)
- application-scoped

Constant Memory

Declared using *__constant__* keyword; the following function is used for copying data to constant memory:

```
cudaError_t cudaMemcpyToSymbol(const char *symbol,
  const void *src, size_t count, size_t offset,
  enum cudaMemcpyKind kind)
```

Data is copied from system memory (*cudaMemcpyHostToDevice*) or global memory (*cudaMemcpyDeviceToDevice*) from *src* into *symbol*. The copied block has *count* bytes. Copied with *offset* into the *symbol* memory.

GPU Local Memory

Texture memory

- cached, 2D locality
- read-only for cache coherency reasons
- high latency
- several addressing modes
  - normalization into $[0, 1]$ range
  - truncation or overflowing of coordinates
- possible data filtering
  - linear interpolation or nearest value
- this functionality is "for free" (implemented in HW)

More details are available in CUDA Programming Guide.

System-Local Memory

System RAM

- connected to GPU using PCIe
- CPU (host) and GPU (device) memory transfers are complicated by virtual addressing
- it is possible to allocate so called page-locked memory areas
    - overall system performance may be reduced
    - limited size
    - data is transferred faster over PCIe
    - allows for parallel kernel run and data copying
    - allows for mapping of host address space onto the device
    - allows for *write-combining* access (data is not cached by CPU)

Page Locked Memory

cudaMallocHost() is used instead of *malloc()* to allocate the memory; the memory is freed using *cudaFreeHost()*

- *cudaHostAllocPortable* flag ensures page-locked memory for all CPU threads
- *cudaHostAllocWriteCombined* flag turns off caching for CPU allocated memory
- *cudaHostAllocMapped* flag sets host memory mapping in the device address space

Page-Locked Memory

Mapped memory

- the same position has a different address for device and host code
- device address may be obtained using *cudaHostGetDevicePointer()*
- before calling any other CUDA API functions, it is necessary to call *cudaSetDeviceFlags()* with *cudaDeviceMapHost* flag

Asynchronous transfers

- API funkce *Async* suffix
- both data transfers – CPU computation and data transfer – GPU computation may be overlapping (more detailed explanation will come with streams)

Non-cached memory

- slow access from host code
- faster access from device memory
- CPU cache doesn't get flushed

Synchronization within the Block

- native barrier synchronization
  - all threads have to enter it (beware of conditions!)
  - one instruction only, very fast if it doesn't degrade parallelism
  - C for CUDA call **__syncthreads()**
  - Fermi extensions: count, and, or
- shared memory communication
  - threads can exchange data
  - synchronization using atomic variables or a barrier

Atomic operations

- performs read-modify-write operations on shared or global memory
- no interference with other threads
- for 32-bit and 64-bit integers (c. c. ≥ 1.2) and float (u c. c. ≥ 2.0)
- using global memory for c. c. ≥ 1.1 and shared memory for c. c. ≥ 1.2
- arithmetic (Add, Sub, Exch, Min, Max, Inc, Dec, CAS) a bitwise (And, Or, Xor) operations

Warp Voting

All threads in one warp evaluate the same condition and perform its comparison. Available in c. c. ≥ 1.2.

```
int __all(int predicate);
```

Result is non-zero iff the predicate is non-zero for all the threads in the warp.

```
int __any(int predicate);
```

Result is non-zero iff the predicate is non-zero for at least one thread in the warp.

```
unsigned int __ballot(int predicate);
```

Contains voting bit mask of individual threads.

Synchronization of Memory Operations

Shared memory is usually used for communication among threads or as a cache for data used by threads.

- threads use data stored by other threads
- it is necessary to ensure that we do not read data that is not available yet
- should we wait, we can use *__syncthreads()*

Synchronization of Memory Operations

Compiler can optimize operations on shared/global memory (intermediate results may be kept in registers) and can reorder them

- if we need to ensure that the data is visible for others, we use *__threadfence()* or *__threadfence_block()*
- if a variable is declared as *volatile*, all load/store operations are implemented in shared/global memory
  - very important if we assume implicit warp synchronization

Block Synchronization

Among blocks

- global memory is visible for all blocks
- poor native support for synchronization
  - no global barrier
  - *atomic operations* on global memory for newer GPUs
  - global barrier can be implemented using kernel calls (another solution is quite tricky)
  - poor options for global synchronization make programming hard but allow for very good scalability

Global Synchronization using Atomic Operations

Problem of sum of elements in a vector

- each block sums elements in its part of a vector
- global barrier
- one block sums results of all the blocks

```
__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float* array, unsigned int N,
  float* result) {
  float partialSum = calculatePartialSum(array, N);
  if (threadIdx.x == 0) {
    result[blockIdx.x] = partialSum;
    __threadfence();
    unsigned int value = atomicInc(&count, gridDim.x);
    isLastBlockDone = (value == (gridDim.x - 1));
  }
  __syncthreads();
  if (isLastBlockDone) {
    float totalSum = calculateTotalSum(result);
    if (threadIdx.x == 0) {
      result[0] = totalSum;
      count = 0;
    }
  }
}
```

Matrix Multiplication

We want to multiply matrices A a B and store the result into C.
For sake of simplicity, we only assume matrices sized $n \times n$.

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$$

C language:

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++){
    C[i*n + j] = 0.0;
    for (int k = 0; k < n; k++)
      C[i*n + j] += A[i*n + k] * B[k*n + j];
  }
```

Parallelization

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++){
    C[i*n + j] = 0.0;
    for (int k = 0; k < n; k++)
      C[i*n + j] += A[i*n + k] * B[k*n + j];
  }
```

Multiple ways of parallelization

- choose one loop
- choose two loops
- parallelize all the loops

Parallelization

Parallelization of one loop

- doesn't scale well, it is necessary to use big matrices (we need thousands of threads for good GPU utilization)

Parallelization

Parallelization of one loop

- doesn't scale well, it is necessary to use big matrices (we need thousands of threads for good GPU utilization)

Parallelization of two loops

- scales well, number of threads grows quadratically w.r.t. n

Parallelization

Parallelization of one loop

- doesn't scale well, it is necessary to use big matrices (we need thousands of threads for good GPU utilization)

Parallelization of two loops

- scales well, number of threads grows quadratically w.r.t. n

Parallelization using inner loop

- bad, synchronization needed when writing into C!

Parallelization

Parallelization of one loop

- doesn't scale well, it is necessary to use big matrices (we need thousands of threads for good GPU utilization)

Parallelization of two loops

- scales well, number of threads grows quadratically w.r.t. n

Parallelization using inner loop

- bad, synchronization needed when writing into C!

Best way is thus to parallelize loops over i and j.

First Kernel

We can form the block and grid as 2D array.

```
__global__ void mmul(float *A, float *B, float *C, int n){
  int x = blockIdx.x*blockDim.x + threadIdx.x;
  int y = blockIdx.y*blockDim.y + threadIdx.y;

  float tmp = 0;
  for (int k = 0; k < n; k++)
    tmp += A[y*n+k] * B[k*n+x];

  C[y*n + x] = tmp;
}
```

Note similarity to math description – parallel version is more intuitive than the serial one!

Performance

What will be the performance of our implementation?

Performance

What will be the performance of our implementation?
Let's look at GeForce GTX 280

- available 622 GFLOPS for matrix multiplication
- memory bandwidth is 142 GB/s

Performance

What will be the performance of our implementation?
Let's look at GeForce GTX 280

- available 622 GFLOPS for matrix multiplication
- memory bandwidth is 142 GB/s

Flop-to-word ratio of our implementation

- in one step over k, we read 2 floats (one number from A and B) and perform two arithmetic operations
- one arithmetic operation corresponds to transfer of one float
- global memory offers throughput of 35.5 billion floats per second if one warp transfers one float from one matrix and 16 floats from the other matrix, we can achieve 66.8 GFLOPS
- 66.8 GFLOPS is very far from 622 GFLOPS

How to Improve It?

We hit the limit of global memory. GPUs have faster types of memory, can we use them?

How to Improve It?

We hit the limit of global memory. GPUs have faster types of memory, can we use them?

For computation of one C element, we have to read one row from A and one column from B, that are in the global memory.

How to Improve It?

We hit the limit of global memory. GPUs have faster types of memory, can we use them?
For computation of one C element, we have to read one row from A and one column from B, that are in the global memory.
Is it really necessary to do that separately for each element of C?

- we read the same A row for all the elements in the same row of C
- we read the same B column for all the elements in the same column of C
- we can read some data only once from the global memory into the shared memory and then read them repeatedly from the shared memory

Blockwise Approach

If we access the matrix in blocks, we can amortize transfers from the global memory:

- we will compute $a \times a$ block of C matrix
- we read blocks of the same size of matrices A and B into the shared memory iteratively
- the blocks will be multiplied and added to C
- arithmetic operations to data transfers is a times better

Natural mapping on GPU parallelism

- individual thread blocks will only compute blocks of C matrix
- they have shared memory
- they get synchronized fast
- no inter-block synchronization needed

Blockwise Access

How big blocks?

- limited by the size of shared memory
- limited by the number of threads that can run on GPU
- if one thread is to compute one element of C, a reasonable block size is 16×16
  - multiple of warp size
  - one block will have reasonable 256 threads
  - one block needs 2 KB of shared memory
  - the memory will not limit the performance ($16 \cdot 25.5 = 568$ GFLOPS, which is quite close to 622 GFLOPS)

Algorithm

Algorithm schema

- each thread block will have *As* and *Bs* in the shared memory
- blocks of *A* and *B* matrices will be multiplied iteratively, the results will get accumulated in *Csub* variable
    - threads in a block read blocks into *As* and *Bs* simultaneously
    - each thread mutliplies blocks in *As* and *Bs* for its element of *Csub* matrix
- each thread stores one element of the matrix into the *C* in global memory

Beware of synchronization

- the blocks need to be read completely before the multiplication starts
- before we read new blocks, operation on previous data needs to be completed

Second Kernel

```
__global__ void mmul(float *A, float *B, float *C, int n){
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
  __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

  float Csub = 0.0f;
  for (int b = 0; b < n/BLOCK_SIZE; b++){
    As[ty][tx] = A[(ty + by*BLOCK_SIZE)*n + b*BLOCK_SIZE+tx];
    Bs[ty][tx] = B[(ty + b*BLOCK_SIZE)*n + bx*BLOCK_SIZE+tx];
    __syncthreads();

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

  C[(ty + by*BLOCK)*n + bx*BLOCK_SIZE+tx] = Csub;
}
```

Performance

- theoretical limitation of first kernel is 66.8 GFLOPS, measured performance is 36.6 GFLOPS
- theoretical limitation of first kernel is 568 GFLOPS, measured performance is 198 GFLOPS
- how to get closer to the maximum performance of the card?
- we need to understand HW and its limitation better and optimize the algorithms accordingly
- topics for the next lecture