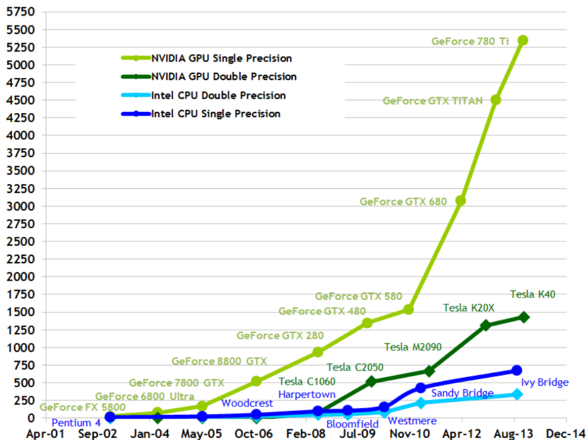
GPU Acceleration of General Computing Tasks

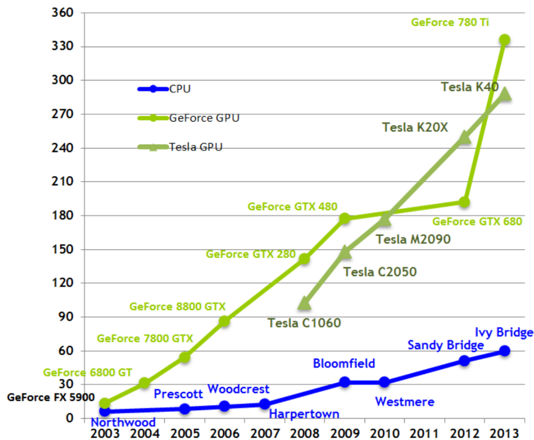
Jiří Filipovič

spring 2023

Motivation – arithmetic performance of GPUs



Motivation – memory bandwidth of GPUs



Motivation – programming complexity

OK, so GPUs are fast, but aren't much more difficult to program?

- well, it's much more complicated than writing serial C++ code...
- but is it fair comparison?

Motivation – programming complexity

OK, so GPUs are fast, but aren't much more difficult to program?

- well, it's much more complicated than writing serial C++ code...
- but is it fair comparison?

Moore's Law

The amount of transistors, which can be placed into single chip, doubles every 18 months

Motivation – programming complexity

OK, so GPUs are fast, but aren't much more difficult to program?

- well, it's much more complicated than writing serial C++ code...
- but is it fair comparison?

Moore's Law

The amount of transistors, which can be placed into single chip, doubles every 18 months

The performance grow is caused by:

- **in the past:** higher frequency, instruction-level parallelism, out-of-order instruction execution, etc.
- **nowadays:** wider vector instructions, more cores

Motivation – the paradigm shift

Consequences of the Moore's Law:

- **in the past:** the changes in processors architectures are relevant for compilers developers
- **nowadays**: we need to explicitly parallelize and vectorize the code to keep scaling the performance
  - still a lot of work for developers, compilers have very limited capabilities here
  - writing of really efficient code is similarly difficult for both GPUs and CPUs

What makes GPU powerful?

Parallelism types

- Task parallelism
  - the problem is decomposed to parallel tasks
  - tasks are typically complex, they can perform different jobs
  - complex synchronization
  - best for lower number of high-performance processors/cores
- Data parallelism
  - the parallelism on a level of data structures
  - typically the same operation on multiple elements of a data structure
  - can be executed on simpler processors

What makes GPU powerful?

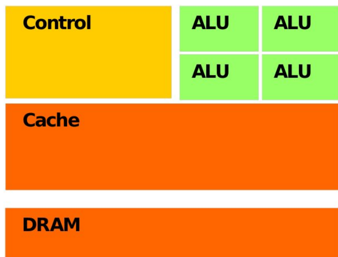
Programmer point of view

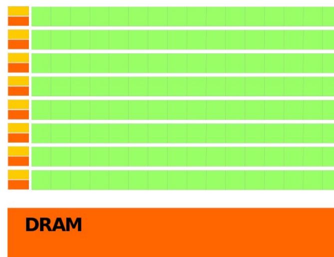
- some problems are more task-parallel, some more data-parallel (tree traversal vs. vector addition)

Hardware designer point of view

- processors for data-parallel computations can be simpler
- so we can get more arithmetic power per square centimeter (i.e., for the same amount of transistors)
- simpler memory access patterns allows to create a memory with higher bandwidth

GPU Architecture





GPU Architecture

CPU vs. GPU

- hundreds ALU in tens of cores vs. tens of thousands ALU in tens of multiprocessors
- out-of-order vs. in-order
- MIMD, SIMD for short vectors vs. SIMT for long vectors
- big cache vs. small cache, often read-only

GPUs use more transistors for ALUs than for cache and instruction control => higher peak performance, less universal

GPU Architecture

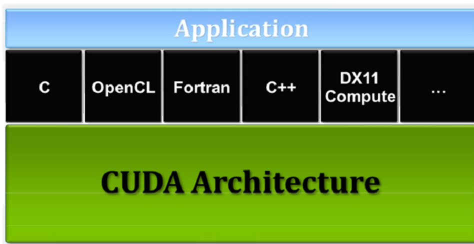
High-end GPU:

- co-processor with dedicated memory
- asynchronous instructions execution
- connected via PCI-E to the rest of the system

CUDA

CUDA (Compute Unified Device Architecture)

- architecture for parallel computations developed by NVIDIA
- a programming model allowing to implement general programs on GPUs
- can be used with multiple programming languages



Processor G80

G80

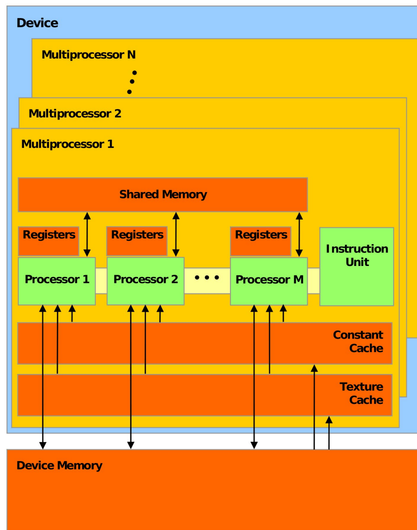
- the first CUDA processor
- contains 16 multiprocessors
- a multiprocessor
  - 8 scalar processors
  - 2 special function units
  - up to 768 threads
    - HW switching and scheduling
  - groups of 32 threads are organized into warps
    - SIMT
  - native synchronization within a multiprocessor

Memory model of G80

Memory model

- 8192 registers shared among all threads within a multiprocessor
- 16 KB shared memory
  - local within a multiprocessor
  - close to the registers' speed (under some circumstances)
- constant memory
  - cached, optimized for broadcast, read-only
- texture memory
  - cached, 2D spatial locality, read-only
- global memory
  - read-write, not cached
- transfers between system and global memory via PCI-E

Processor G80



Newer GPUs

Similar architecture, new features

- double-precision
- relaxed rules for efficient access into global memory
- L1, L2/data cache
- higher amount of on-chip resources (registers, shared memory, threads etc.)
- wider synchronization options (e.g., atomic operations)
- nested parallelism
- unified memory

C for CUDA

C for CUDA extends C/C++ language for parallel computations with GPUs

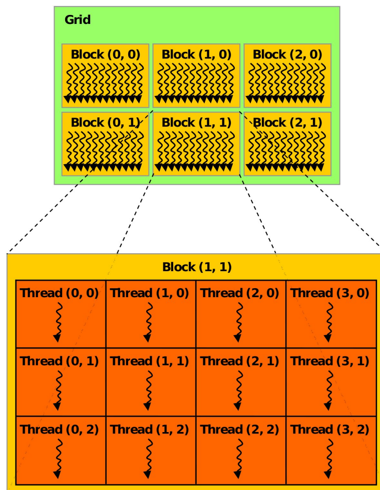
- explicit separation of a host (CPU) and a device (GPU) code
- threads hierarchy
- memory hierarchy
- synchronization mechanisms
- API (context manipulation, memory, errors handling etc.)

Threads hierarchy

Threads hierarchy

- threads are organized into thread-blocks
- thread-blocks create a grid
- a computational problem is typically decomposed into independent sub-problems, solved by thread-blocks
- subproblems are further parallelized and solved by (potentially collaborating) threads
- ensures good scaling

Threads hierarchy

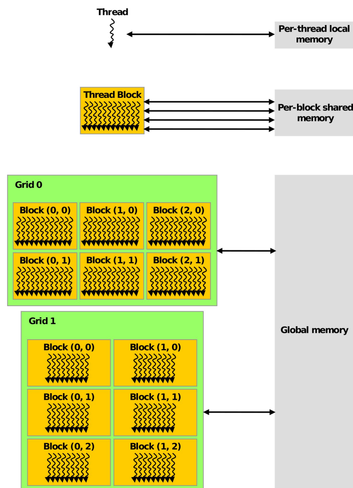


Memory hierarchy

Multiple types of memory

- differ in visibility
- differ in life-time
- differ in latency and bandwidth

Memory hierarchy



Example – vector addition

We want to add vectors a, b, and store the result into vector c.

Example – vector addition

We want to add vectors a, b, and store the result into vector c.
We need to parallelize the problem.

Example – vector addition

We want to add vectors *a*, *b*, and store the result into vector *c*.

We need to parallelize the problem.

Serial code:

```
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

Example – vector addition

We want to add vectors a, b, and store the result into vector c.

We need to parallelize the problem.

Serial code:

```
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

Independent iterations – easy to parallelize, scales with the vector size.

Example – vector addition

We want to add vectors a, b, and store the result into vector c.
We need to parallelize the problem.
Serial code:

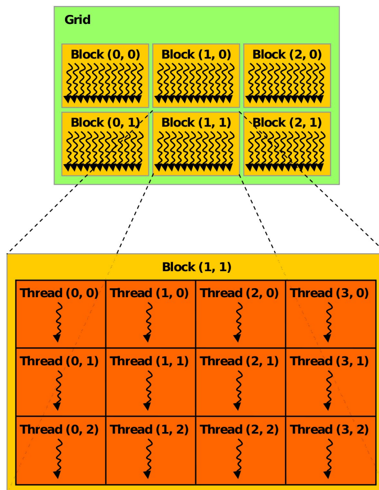
```
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

Independent iterations – easy to parallelize, scales with the vector size.
i-th thread adds i-th elements of a, b:

```
c[i] = a[i] + b[i];
```

How to find out which index to pick?

Threads hierarchy



Identification of a thread and a block

Each thread in C for CUDA has build-in variables:

- **threadIdx.**{**x, y, z**} contains the position of the thread within its block
- **blockDim.**{**x, y, z**} contains the size of the block
- **blockIdx.**{**x, y, z**} contains the position of the block within a grid
- **gridDim.**{**x, y, z**} contains the size of the grid

Example – vector addition

We need to compute a global position of the thread (using 1D blocks and grid):

Example – vector addition

We need to compute a global position of the thread (using 1D blocks and grid):

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

Example – vector addition

We need to compute a global position of the thread (using 1D blocks and grid):

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

The complete function for the parallel vector addition:

```
__global__ void addvec(float *a, float *b, float *c){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}
```

Example – vector addition

We need to compute a global position of the thread (using 1D blocks and grid):

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

The complete function for the parallel vector addition:

```
__global__ void addvec(float *a, float *b, float *c){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}
```

The code defines a kernel (a parallel function executed on GPU).

When executing kernel, the size of block and number of blocks has to be defined.

Function type quantifiers

The syntax of C is extended by function type quantifiers, determining from where the function can be called and where it is executed

- **__device__** function is executed on device (GPU) and called from device code
- **__global__** function is executed on device and called from host (CPU)
- **__host__** function is executed on host, and called from host
- `__host__` and `__device__` can be combined, the function is then compiled for both host and device and also can be called from both host and device

Example – vector addition

For complete computation of vector addition, we need to:

Example – vector addition

For complete computation of vector addition, we need to:

- allocate memory for the vectors, and fill it with some data

Example – vector addition

For complete computation of vector addition, we need to:

- allocate memory for the vectors, and fill it with some data
- *allocate GPU memory*

Example – vector addition

For complete computation of vector addition, we need to:

- allocate memory for the vectors, and fill it with some data
- *allocate GPU memory*
- *copy vectors a a b to GPU memory*

Example – vector addition

For complete computation of vector addition, we need to:

- allocate memory for the vectors, and fill it with some data
- *allocate GPU memory*
- *copy vectors a a b to GPU memory*
- **compute vector addition on GPU**

Example – vector addition

For complete computation of vector addition, we need to:

- allocate memory for the vectors, and fill it with some data
- *allocate GPU memory*
- *copy vectors a a b to GPU memory*
- **compute vector addition on GPU**
- *copy back the result from GPU memory into c*

Example – vector addition

For complete computation of vector addition, we need to:

- allocate memory for the vectors, and fill it with some data
- *allocate GPU memory*
- *copy vectors a a b to GPU memory*
- **compute vector addition on GPU**
- *copy back the result from GPU memory into c*
- use *c* somehow :-)

When managed memory is used (supported from compute capability 3.0 and CUDA 6.0), we don't need to perform steps printed in *italic*.

Example – vector addition

CPU code fills a a b, and prints c:

```
#include <stdio.h>
#define N 64
int main(){
  float *a, *b, *c;
  cudaMallocManaged(&a, N*sizeof(*a));
  cudaMallocManaged(&b, N*sizeof(*b));
  cudaMallocManaged(&c, N*sizeof(*c));

  for (int i = 0; i < N; i++) {
    a[i] = i; b[i] = i*2; }

// placeholder for GPU computation

  for (int i = 0; i < N; i++)
    printf("%f, ", c[i]);

  cudaFree(a); cudaFree(b); cudaFree(c);

  return 0;
}
```

GPU memory management

We use managed memory, so CUDA automatically copies data between CPU and GPU.

- memory coherency is automatically ensured
- we cannot access managed memory while any GPU kernel is running (even if it does not touch the buffer we want to use)

Alternatively, we can allocate and copy memory explicitly:

```
cudaMalloc(void** devPtr, size_t count);
cudaFree(void* devPtr);
cudaMemcpy(void* dst, const void* src, size_t count,
  enum cudaMemcpyKind kind);
```

Example – vector addition

Kernel execution:

- the kernel is called as a C-function; between the name and the arguments, there are triple angle brackets with specification of grid and block size
- we need to know block size and their count
- we will use 1D block and grid with fixed block size
- the size of the grid is determined in a way to compute the whole problem of vector sum

For vector size divisible by 32:

```
#define BLOCK 32
addvec<<<N/BLOCK, BLOCK>>>(a, b, c);
cudaDeviceSynchronize();
```

The synchronization after kernel call ensures that c is going to be accessed by host code after the called kernel finishes.

Example – vector addition

How to solve a general vector size?
We will modify the kernel source:

```
__global__ void addvec(float *a, float *b, float *c, int n){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) c[i] = a[i] + b[i];
}
```

And call the kernel with sufficient number of threads:

```
addvec<<<N/BLOCK + 1, BLOCK>>>(a, b, c, N);
```

Compilation

Now we just need to compile it :-).

```
nvcc -o vecadd vecadd.cu
```

Thread-local memory

Registers

- the fastest memory, directly used by instructions
- local variables and intermediate results are stored into registers
  - if there is enough registers
  - if compiler can determine array indexes in compile time
- life-time of a thread

Thread-local memory

Registers

- the fastest memory, directly used by instructions
- local variables and intermediate results are stored into registers
  - if there is enough registers
  - if compiler can determine array indexes in compile time
- life-time of a thread

Local memory

- what cannot fit into registers, goes to the local memory
- physically stored in global memory, have longer latency and lower bandwidth
- life-time of a thread

Block-local memory

Shared memory

- the speed is close to registers
  - if there are no bank-conflicts
  - typically requires some load/store instructions
- declared by __*shared*__
- can have dynamic size (determined during kernel execution), if declared as *extern* without specification of the array size
- life-time of a thread block

GPU-local memory

Global memory

- order-of-magnitude lower bandwidth compared to the shared memory
- latency in hundreds of GPU clocks
- coalesced access necessary for efficient access
- life-time of an application
- can be cached (depending on GPU architecture)

Dynamic allocation with *cudaMalloc*, static allocation by using *__device__*

Other memories

- constant memory
- texture memory
- system memory

Thread block-scope synchronization

- native barrier
  - has to be visited by all threads within a thread-block
  - only one instruction, very fast if not reduce parallelism
  - **__syncthreads()**

Atomic operations

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

Synchronization of memory operations

Compiler can optimize access into shared and global memory by placing intermediate results into registers, and it can change order of memory operations:

- *__threadfence()* and *__threadfence_block()* can be used to ensure data we are storing are visible for others
- variables declared as *volatile* are always read/written from/to global or shared memory

Thread-block synchronization

Thread blocks communication

- global memory visible for all blocks
- but weak possibilities to synchronize between blocks
  - in general no global barrier (can be implemented if all blocks are persistent on GPU)
  - using atomic operations can solve some problems
  - generic global barrier only by kernel invocation
  - harder to program, but allows better scaling

Global synchronization via atomic operations

Alternative implementation of vector reduction

- each thread-block reduces a subvector
- the last running thread-block adds results of all thread-blocks
  - implementation of weak global barrier: after finishing blocks $1..n-1$, blocks n continues

```
__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;
    }
  }
}
```

Materials

CUDA documentation (part of CUDA Toolkit, downloadable from *developer.nvidia.com*)

- CUDA C Programming Guide (CUDA essentials)
- CUDA C Best Practices Guide (more details on optimization)
- CUDA Reference Manual (complete C for CUDA API reference)
- a lot of other useful documents (nvcc manual, documentation of PTX and assembly, documentation for various accelerated libraries, etc.)

CUDA, Supercomputing for the Masses

- http://www.ddj.com/cpp/207200659

Today, we learned

- what is CUDA good for
- basic GPU architecture
- basic C for CUDA programming

In the next lecture, we will focus

- how to write efficient GPU code