General Purpose GPU Programming

Petr Baudiš ⟨pasky@suse.cz⟩

SUSE Labs Conf 2010

Motivation

- Commodity computers today have *another* pretty powerful computer inside — underused!
- Theoretical possibility: 100$\times$ speedup top commodity GPU vs. top x86 CPU
- It is pretty easy to code for it, but efficient code can be very tricky
  - It is difficult to parallelize most algorithms suitably
  - High latency — you should work on a lot of data
  - A lot of device quirks — scheduling, memory latency, ...
- Remember IBM Cell?

Motivation

- Commodity computers today have *another* pretty powerful computer inside — underused!
- Theoretical possibility: $100\times$ speedup top commodity GPU vs. top x86 CPU
- It is pretty easy to code for it, but efficient code can be very tricky
  - It is difficult to parallelize most algorithms suitably
  - High latency — you should work on a lot of data
  - A lot of device quirks — scheduling, memory latency, ...
- Remember IBM Cell?
- I'm not an expert!
- Focus on NVidia

What will we talk about

- GPU — what it is, how it works, what it can and cannot do
- GPU Programming Tools
- GPU Programming Concepts
- Few Examples

Outline

GPU: A History

- (80s) Amiga — The first computer with a (2D) graphics accelerator

GPU: A History

- (80s) Amiga — The first computer with a (2D) graphics accelerator
- (1996) 3dfx Voodoo — The first mass-available 3D accelerator (everything hardcoded)

GPU: A History

- (80s) Amiga — The first computer with a (2D) graphics accelerator
- (1996) 3dfx Voodoo — The first mass-available 3D accelerator (everything hardcoded)
- (2002) NV20, R300 — The first GPUs with programmable vertex, fragment shaders
- (2006) G80, R600 — Unified shader architecture: fully programmable units

GPU: A History

- (80s) Amiga — The first computer with a (2D) graphics accelerator
- (1996) 3dfx Voodoo — The first mass-available 3D accelerator (everything hardcoded)
- (2002) NV20, R300 — The first GPUs with programmable vertex, fragment shaders
- (2006) G80, R600 — Unified shader architecture: fully programmable units
- (2006) AMD FireStream, (2008) NVidia Tesla — "GPUs" without video output

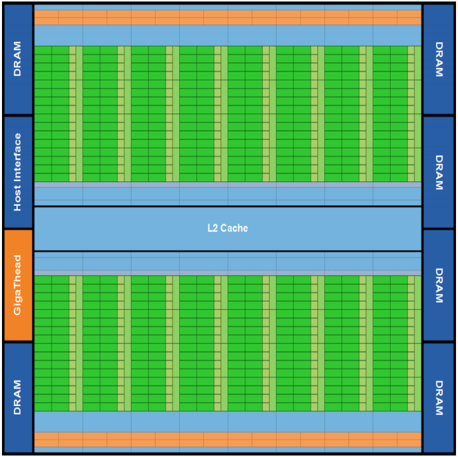
GPU Architecture

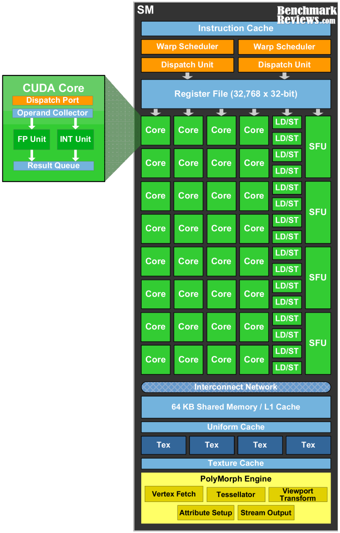
- **On-board Memory** is pretty fast and pretty large, but has latency; small L2 cache
- **Multiprocessors** talk to memory and execure “simple” programs *(shader kernels)* on many cores, have some small local memory
- **Cores** are SIMT computational units — they must all execute single instruction at once! (If one core needs to diverge, all others are masked out and just wait)
- **Instruction set** is reasonable, Turing-complete, can do fast ops with both ints and floats
- **Register file** is huge ($\times$ many threads share a core, all are local variables)

GPU Architecture

- **On-board Memory** is pretty fast and pretty large, but has latency; small L2 cache
- **Multiprocessors** talk to memory and execure "simple" programs *(shader kernels)* on many cores, have some small local memory
- **Cores** are SIMT computational units — they must all execute single instruction at once! (If one core needs to diverge, all others are masked out and just wait)
- **Instruction set** is reasonable, Turing-complete, can do fast ops with both ints and floats
- **Register file** is huge ($\times$ many threads share a core, all are local variables)
- **ATI Perspective:** Much less cores than NVidia, but each core is SIMD: 5-element vector unit

GPU Block Diagram





Concrete Devices

GeForce GTX 260

Compute capability:	1.3
Total amount of global memory:	938803200 B
Number of multiprocessors:	24
Number of cores:	192
Total amount of constant memory:	65536 B
Total amount of shared memory per block:	16384 B
Total number of reg. available per block:	16384
Warp size:	32
Maximum number of threads per block:	512
Clock rate:	1.24 GHz

- Relatively good gaming GPU
- Commodity GPU almost the same, but 2 multiprocessors
- Fermi (GT100): Compute Capability 2.0, $32 \times 16 = 512$ cores

Outline

Past Programming Tools

Assembler

- Hard-core, device-specific
- Actually documented!

Past Programming Tools

Assembler

- Hard-core, device-specific
- Actually documented!

OpenGL (GLSL), DirectX (HLSL)

- Historical, still useful for generating complex graphics (e.g. fractals); C-like syntax
- Limited capabilities, clumsy, but partially device-portable

Past Programming Tools

Assembler

- Hard-core, device-specific
- Actually documented!

OpenGL (GLSL), DirectX (HLSL)

- Historical, still useful for generating complex graphics (e.g. fractals); C-like syntax
- Limited capabilities, clumsy, but partially device-portable

AMD Stream Close-to-Metal

- Historical, similar to CUDA, C-like syntax
- Mostly discontinued in favor of OpenCL
- fglrx driver

NVidia CUDA

- Compute Unified Device Architecture
- Completely NVidia-specific, but can use all the features (Compute Capability levels)
- Nice SDK, big base of existing applications and examples
- Contains some debugging, profiling tools, CPU emulation
- C-like syntax, gets compiled to `.o` and linked to your program
- Will probably be phased out later, but still used a lot
- nv driver

OpenCL

- Open Computing Language
- Initiated by Apple, maintained by Khronos, all vendors contribute
- Support for CPUs, GPUs, Cell, ...
- Still quite young technology, C99 extension but compilation on runtime
- Young drivers, not too stable; nv, fglrx

Outline

Programming Concepts

- We can execute several *workgroups (blocks)* on the GPU on different multiprocessors
- We execute single workgroup *kernel* on the GPU in many *threads*

Programming Concepts

- We can execute several *workgroups (blocks)* on the GPU on different multiprocessors
- We execute single workgroup *kernel* on the GPU in many *threads*
- We divide the set of threads to *warps* — threads in a warp run concurrently
- When threads in the warp request memory, another warp of the workgroup gets scheduled
- Frequently, warp size $>$ core count: n-pumped cores (NVidia)

Special Operations

Special Registers

- Each thread can gets its id within the workgroup (block)
- Each thread can get an id of the workgroup (block)

Special Operations

- Single-instruction mathematical functions (e.g. sqrt, sin, log)
- Asynchronous memory transfer
- Synchronization: Some fences and barriers, typically only within single block; single-warp atomic instructions and voting

Variables

Available Types

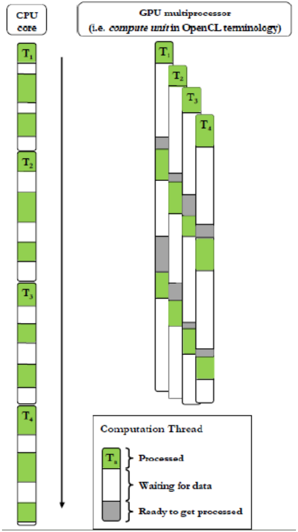
- `char`, `int`, `float`
- `double` is slower on older devices, precision tradeoffs
- `int4`, `int`*n*, `float`*n* — vectors
- Fast swizzling supported: `float4 dup = vec.xxyy, rot = vec.yzwx;`

Storage Classes

- Global — on-GPU memory shared by all blocks (slow)
- Local — on-GPU memory reserved for current block (slow!)
- Texture — piece of global memory with fast random access
- Shared — on-multiprocessor memory for current block (small)
- Auto — all local variables are in registers

Memory Latency Hiding

- On CPU, context switch is expensive, want big cache
- On GPU, context switch is cheap, small cache is ok!
- Memory can still stall if access is not coalesced:
  - Compute Capability-dependent; newer cards have more relaxed requirements
  - Older devices need precisely in-sequence accesses within the warp
  - Newer devices coalesce all accesses in-warp; threads accessing single memory segment still means less requests, obviously



Design Patterns

Conditions and loops

- Branching is bad — some cores are idle when others branch!
  - Bad (ex!): `if (up) y += dy; else y -= dy;`
  - Good: `int f = up ? 1 : -1; y += f * dy;`
- `for`-loops unrolled, `while`-loops problematic
- Recursion not supported — no real stack! (Everything inlined.)

Dividing work

- Approach 1: Many instances of a task, each thread solves one
- Approach 2: Many instances of a task, each block solves one
- Approach 3: Single instance of a task, all threads in all blocks cooperate
- Threads divide input dataset into blocks, data dependency problems

Outline

Bitmap Operation — One Bitmap per Block

```
__device__ void invbyte(uchar *bitmap, int c) {
  bitmap[c] ^= ~0;
}
__global__ void invthread(uchar *gbitmap, int sz) {
  uchar *bitmap = &gbitmap[gridDim.x * blockIdx.x];
  // Divide the image to many segments; within one
  // segment, each thread will flip one byte.
  int segsz = blockDim.x;
  for (int i = 0; i < sz; i += segsz) {
    inverse_byte(bitmap, i + threadIdx.x);
  }
}
int main(void) {
  cudaMemcpy(..., ..., i, cudaMemcpyHostToDevice);
  invthread <<<bks, thrs>>> (bitmaps, size);
  cudaMemcpy(..., ..., i, cudaMemcpyDevicetoHost);
}
```

Bitmap Operation — One Bitmap for All

```
__global__ void invthread(uchar *bitmap, int sz) {
  // Divide the picture to many blocks; threads
  // on one multiprocessor work in that block.
  int blksz = sz / gridDim.x;

  // Divide the block to many segments; within one
  // segment, each thread will flip one byte.
  int segsz = blockDim.x;
  for (int i = 0; i < blksz; i += segsz) {
    bitmap[i + threadIdx.x] ^= ~0;
  }
}
```

Matrix Multiplication

```
__global__ void mul_matrix (const float *m1,
    const float *m2, float *mRes) {
  int n = blockDim.x;
  int r = threadDim.x;
  int c = threadDim.y;

  float sum = 0;
  for (int i = 0; i < n; ++i)
    sum += m1[r*n + i] * m2[c*n + i];

  mRes[r*n + c] = sum;
}
```

Matrix Multiplication Caveats

```
__global__ void mul_matrix (const float *m1,
    const float *m2, float *mRes) {
  int n = blockDim.x;
  int r = threadDim.x;
  int c = threadDim.y;

  float sum = 0;
  for (int i = 0; i < n; ++i)
    // We assume m2[] is transposed
    // Totally uncoalesced! Divide to tiles
    // and pre-load to shared memory
    sum += m1[r*n + i] * m2[c*n + i];

  mRes[r*n + c] = sum;
}
```

Regex Matching on Packets

- Parse PCRE string to DFA (Deterministic Finite Automaton) state transition table on the host
- Buffering DFA-tagged packets in page-locked memory
- Periodically transferring packets to the device
- Computation is one thread (walking through the DFA) per packet
- Result is one byte per packet
- `http://www.eecs.ucf.edu/ zhou/pldi10.pdf`

Remember...

- GFLOPs to TFLOPs of computing capability, BUT...
- High host-device latency
- All threads within a block should execute the same instruction at one time
- CUDA $\rightarrow$ OpenCL transation ongoing now

Thank you!

Thank you! Final questions?

Resources

- Many projects, tutorials, forums: http://gpgpu.org/
- CUDA Programming Guide (very good resource!)
- CUDA SDK (huge body of examples)
- OpenCL Specs
- http://www.microway.com/pdfs/GPGPU_Architecture_and_Performance_Comparison.pdf

Figures (c) NVidia