General-purpose computing on graphics processing units (GPGPU)

Thomas Ægidiussen Jensen
Henrik Anker Rasmussen
François Rosé

November 1, 2010

Table of Contents

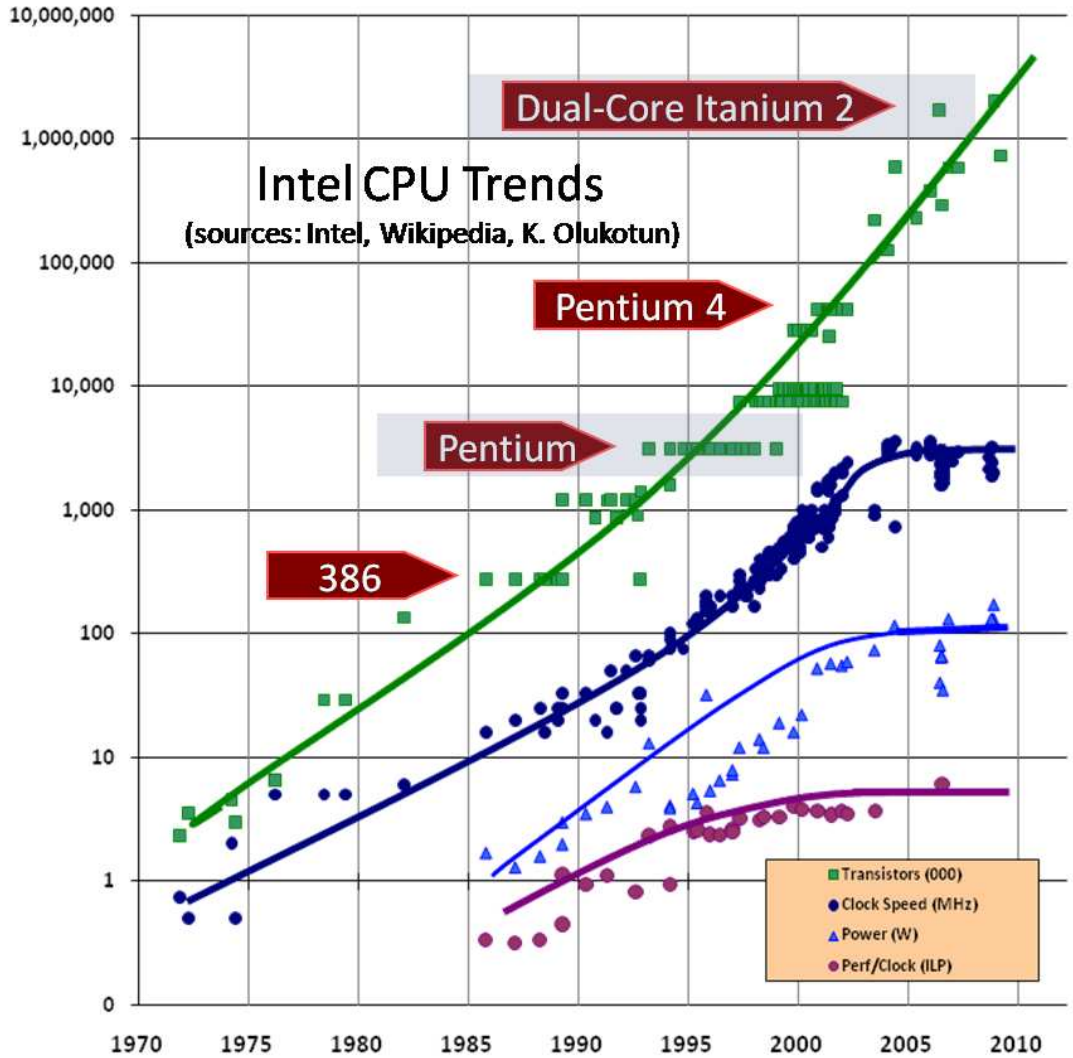
Motivation

- Humans want more!
- Increasing demands for computational power
  - Simulations (fluid, physics, etc.)
  - Entertainment (game, movies, etc.)

However:

- The limit of CPU frequency is reached
- The limit of power consumption is reached

Motivation (cont'd)



Motivation (cont'd)

- We just start using multi-processors
- or multi-cores
  - AMD Phenom II X2
  - Intel Core Duo
- Why not use many cores?

Fortunately, it has already been done
- and is already consumer prized

Graphics processing unit (GPU)

- A specialized hardware for 3D and 2D graphics rendering

3D rendering operations:

- Floating point operations
- Matrix/vector calculations

As a consequence of this:

- A GPU is highly parallelized
- Optimized for floating point operations

Graphics processing unit (GPU) (cont'd)

Figure: CPU vs GPU

- More transistors used for data processing
- Less transistors used for caching and flow control

GPGPU

- A GPU's properties makes it suited for other things than graphics
  - **Image processing**
  - Physics simulation
  - Fluid simulation
  - Ray-tracing (light simulation)
  - ...anything suited to be calculated in parallel

This is what General-Purpose computing on Graphics Processing Unit is all about

Examples of GPGPU Usage - FastHOG

- Library developed by Prisacariu and Reid from Oxford
- HOG: Histogram of Oriented Gradients
- Used for human detection
- Used in our project
- 67x speed improvement
  - Using two GeForce GTX 285

Examples of GPGPU Usage - GSM decrypting

- Karsten Nohl: German GSM expert
- Recorded a GSM phone call using "GSM catcher"
- Crack the key using ATI graphics card in 30 seconds
- Using a rainbow table
- Was able to decode the call and play it

Programming GPGPU

Three widely used programming APIs

- Firestream (AMD)
- **Cuda (nVidia)**
- OpenCL (Khronos Group)

Facts About CUDA

- A short for Compute Unified Device Architecture
- Introduced in November 2006 by nVidia
- GPGPU API for nVidia graphic cards
- Supports all GFX cards with G80 GPUs or better
- Compute capability is a expression for how good a device is for CUDA
- Looks like C/C++
- SDKs for Windows, Linux and Mac OS X
- Scalable programming model

CUDA - Kernels

- Kernels are CUDA functions
- Defined like normal C functions
- when called,
  - Executed on the GPU
  - Started N times
  - In N separate threads
- Built in variable to identify which thread it is executed as
- Used to manage the dividing of a task

CUDA - Blocks

- Threads are grouped into M blocks
- Organized as 1D, 2D or 3D
- Max 1024 threads in each block
- Threads within a block can cooperate through shared memory
- Threads have built-in synchronization functions

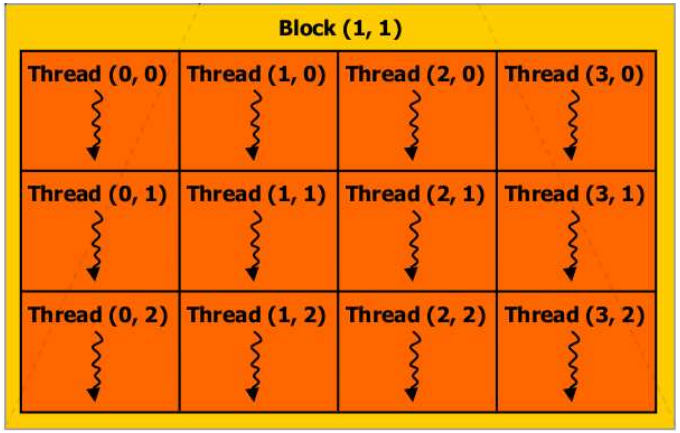


Figure: Block of threads

CUDA - Thread Hierarchy

- Blocks are grouped into a grid
- Organized as 1D or 2D
- A block cannot depend on other blocks
- In practice, a block is scheduled and executed in warps of 32 threads.

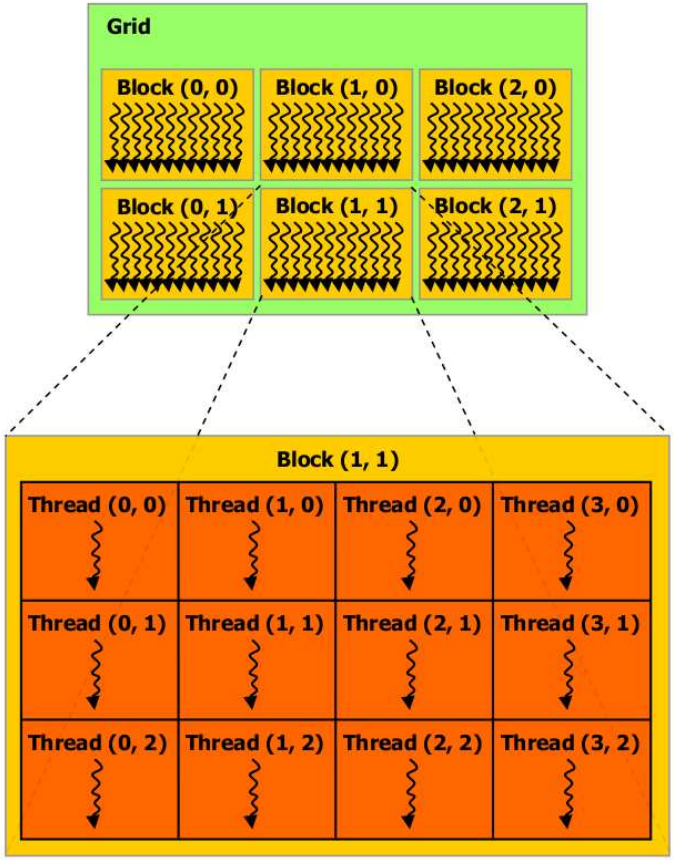
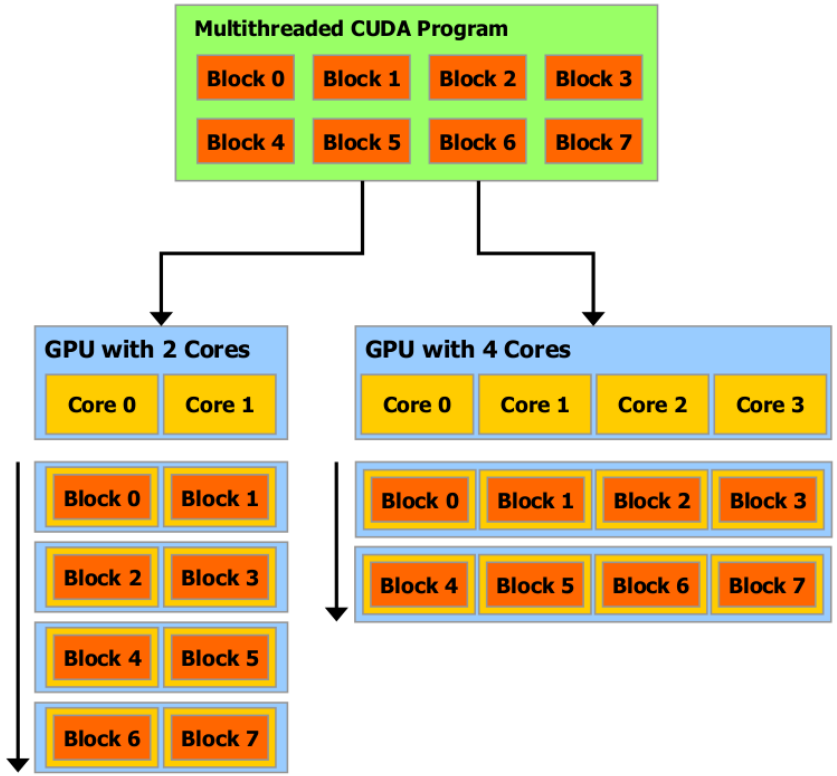


Figure: Thread Hierarchy

CUDA - Scalability

- Encourage to split the tasks into smaller parts called blocks
- Each block can be executed in any order



CUDA - Memory Hierarchy

- Each thread have some local memory
- The threads within a block have shared memory
- Close to the cores and therefore fast memory
- Shared memory lasts as long as the block

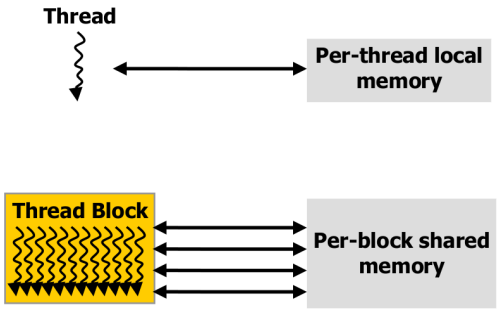


Figure: Thread and block memory hierarchy

CUDA - Memory Hierarchy

- All threads have access to the global memory space
- The threads have two read only memory spaces
  - Constant memory
  - Texture memory
- The transfer from host memory to the global memory is slow

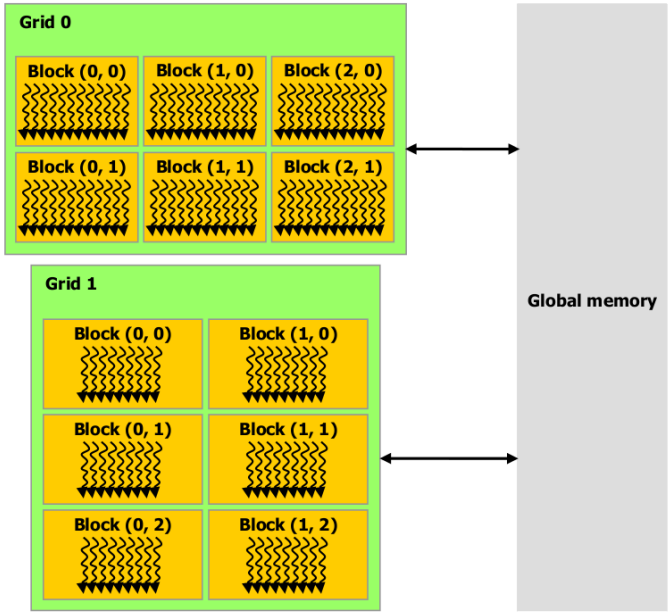


Figure: Global memory

CUDA Programming

Similar to C/C++, although some restrictions

- Only GPU memory can be accessed
- No variable number of arguments
- No static variables
- No recursion
- No dynamic polymorphism

CUDA Programming (cont'd)

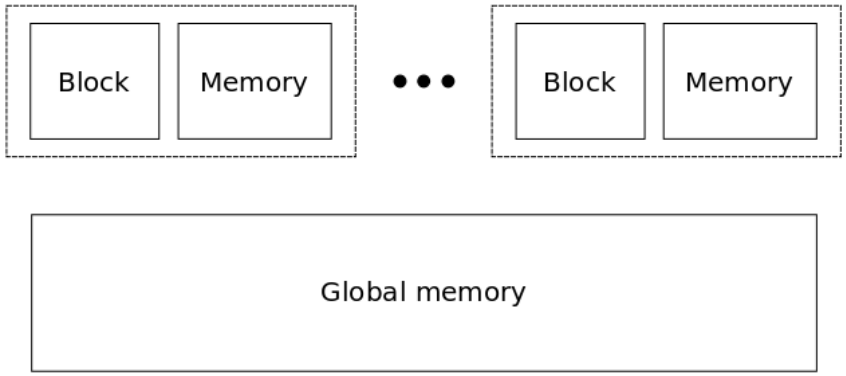
Main flow

1. Data is allocated and created on host (CPU) memory
2. Device (GPU) memory is allocated
3. Data is transferred from host to device memory
4. Data is processed in many parallel threads on device
   - Kernels launched in 1,000s or 10,000s of threads
5. Results are transferred back to host and data is freed

CUDA Programming (cont'd)

Memory types

- Shared memory
  - Threads within same block can cooperate
- Global memory
  - Accessible by all threads



CUDA Programming (cont'd)

Built-in variables and functions in device code

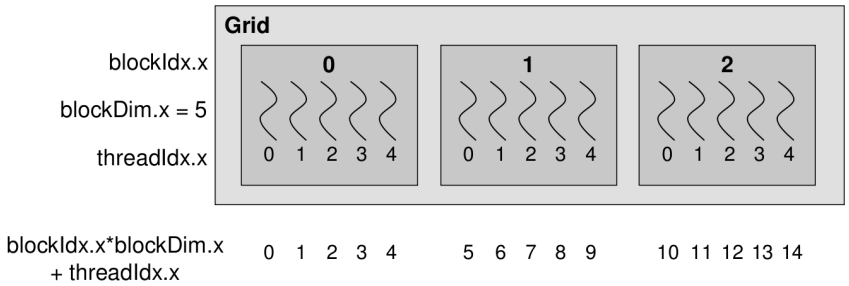
```
dim3 gridDim;              // Grid dimension
dim3 blockDim;             // Block dimension
dim3 blockIdx;             // Block index
dim3 threadIdx;            // Thread index
void __synchthreads();     // Thread synch within block
```

Launching a kernel

```
dim3 grid_dim(100,50); // 5000 thread blocks
dim3 block_dim(4,8,8); // 256 threads per block
my_kernel <<< grid_dim, block_dim >>> (data);
```

CUDA Programming (cont'd)

Example of thread access



CUDA Programming (cont'd)

Indexes are multi-dimensional

- Block: 1D or 2D
- Thread: 1D, 2D or 3D
- Simplifies memory addressing when processing multi-dimensional data
  - Image processing

CUDA Programming (cont'd)

- Kernels are declared with a qualifier
  - __global__ : launched by CPU

    (cannot be called from GPU)
  - __device__ : called from other GPU functions

    (cannot be called by the CPU)
  - __host__ : can be called by CPU
    (__host__ and __device__ qualifiers can be combined)

CUDA Programming (cont'd)

Simplified code (int data)

```
int main()
{
  memSize = numBlocks*numThreadsPerBlock*sizeof(int);

  h_a = (int*)malloc(memSize);
  cudaMalloc(&d_a, memSize);

  cudaMemcpy(d_a, h_a, memSize, cudaMemcpyHostToDevice);
  myKernel<<<grid_dim, block_dim>>>(d_b, d_a);
  cudaMemcpy(h_a, d_b, memSize, cudaMemcpyDeviceToHost);
}
```

Simple CUDA Kernel

```
__global__ void v_add(float* A, float* B, float* C)
{
 int i = threadIdx.x + blockDim.x * blockIdx.x;
 C[i] = A[i] + B[i];
}

int main()
{
 // Allocation and initialization code
 ...
 // Launch kernel
 v_add<<<grid_dim,block_dim>>>(d_A,d_B,d_C);

 // Result is transferred to host and data is freed
 ...}
```

Optimization

Various simple methods for optimizing CUDA code

- Minimize transfers between host and device
  - 4GB/s peak PCIe bandwith vs. 76GB/s peak (Tesla C870)
- Group transfers
  - One large transfer better than many small
- Shared memory is $\sim$100x faster than global memory
  - Cache data to reduce global data access
- Maximize global memory bandwich
  - Coalescing - maximizing throughput (some restrictions)

Often rather do extra computations on GPU to avoid transfers

Break

More optimization after the break!

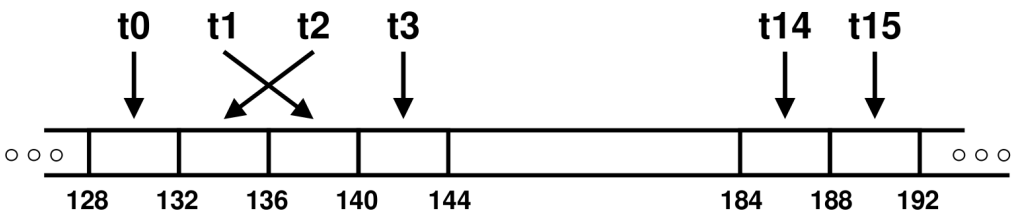
Optimization (cont'd)

Coalescing

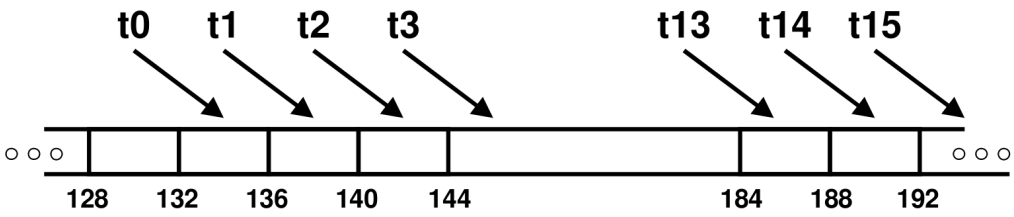
- Minimize number of bus transactions
- A coordinated read by a half-warp (16 threads)
- The half-warp must fulfill some requirements
- Different requirements for different compute capabilities

Optimization (cont'd)

- Uncoalesced access



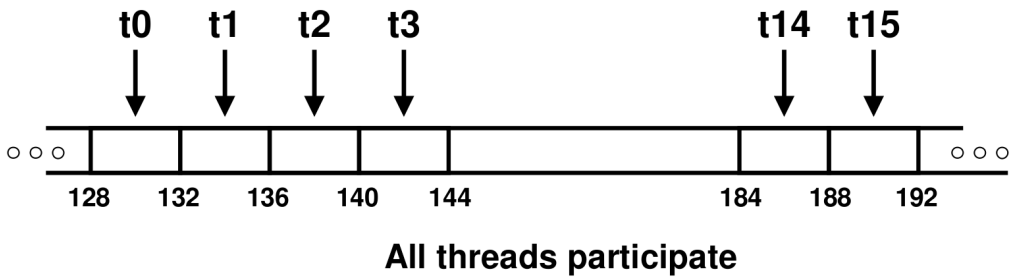
Permuted Access by Threads

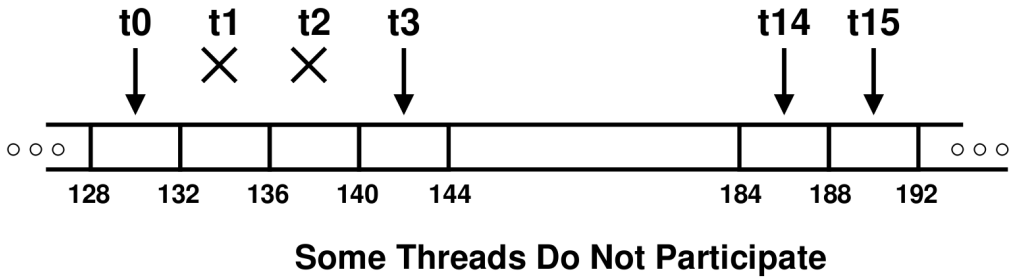


Misaligned Starting Address (not a multiple of 64)

Optimization (cont'd)

- Coalesced access





Optimization (cont'd)

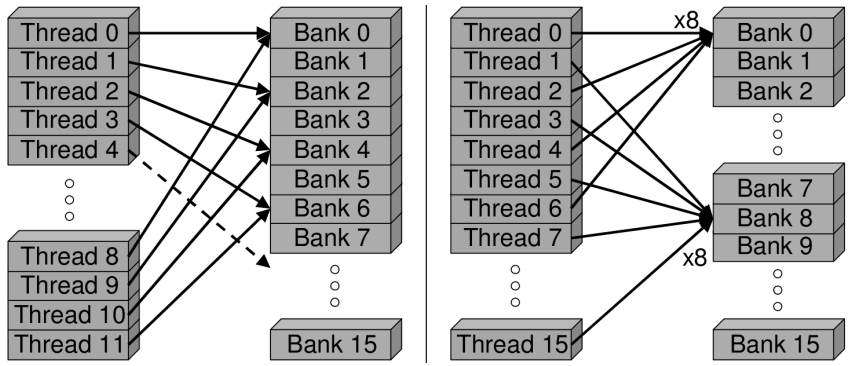
Coalescing

- Compute capability $<$1.2
  - Size of the words read by threads must be 4, 8 or 16B
  - Global memory region must be contigous and 64, 128 or 256B
  - Threads must access words in a sequence
    - The k^{th} thread in the half-warp must access the k^{th} word
    - Although not all threads have to participate
- Compute capability 1.2 and 1.3
  - Almost the same requirements
  - Threads can access words in any order
- Compute capability 2.x ommitted

Optimization (cont'd)

Shared memory should be used as much as possible

- Reduce slow host-device transfers
- Avoids non-coalesced access
- Memory is divided into banks
  - Each bank can service one address per cycle
  - Bank conflict: Multiple simoultaneous access to the same bank
  - Workarounds



Optimization (cont'd)

Keep the hardware busy to hide latencies

- # of blocks > # of multiprocessors
  - All multiprocessors should have at least one block to execute
- # of blocks / # of multiprocessors > 2
  - Multiple blocks can run at the same time on a multiprocessor
- # of blocks > 100
  - Scale to future devices
  - 1,000 should scale across multiple generations

Optimization (cont'd)

Number of threads per block

- Rules of thumb
  - Should be a multiple of warp size (32)
  - Minimum is 64 threads per block
  - 192 or 256 is usually the best choice
  - Too many threads can cause kernel launches to fail (not enough registers per thread)
- Depends on application and HW
  - Experiments might be necessary

Optimization (cont'd)

Summary

- Make efficient parallelism
- Use shared memory as much as possible
- Avoid (reduce) bank conflicts
- Use coalesced memory access
- Make use of other memory spaces (texture, constant)

Kernel Performance

Results of optimization

- A kernel that reads a float, increments it and writes back
  - Uncoalesced access: $3,494\mu s$
  - Coalesced: $356\mu s$
- "Performance for 4M element reduction"
  - parallel reduction by kernel decomposition
  (CUDA Technical Training Volume II, Nvidia)

	Time (2^{22} ints)	Bandwidth	Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s	
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x

Demo

Future of GPGPU

- Early efforts: GPGPU for its own sake
  - Challenge of achieving non-graphics computation on GPU
  - No effort on well-optimized CPU analogs
- The bar is now high
  - Go beyond simple porting
  - Demonstrate general principles and techniques
  - Find new uses of the hardware
- Emergence of high-level languages
  - Leap forward for GPU developers
  - Same for non-graphics developers

Future of GPGPU (cont'd)

- Current and future uses
  - The search for extra-terrestrial intelligence (SETI)
  - Accelerated encryption, compression
  - Accelerated interconversion of video file formats
- First generation of data-parallel coprocessors
  - Rapid growth curve
  - Advantages of parallel computing
- Find the right balance between:
  - Improved generality
  - Increasing performance
- New and interesting opportunities
  - Sony Playstation 3
  - Cell processor + GPU with high-bandwidth bus

CPU/GPU Hybrid

- What is AMD Fusion?
  - Merger of AMD and ATI
  - Integrated CPU and GPU in one unit = APU
  - Multi-core microprocessor architecture
  - Different clocks for graphics and central processing cores
- Llano specifications
  - DirectX 11 compliant
  - PCIe 2.0 controller
  - DDR3-1600 memory controller
  - 1MB L2 cache per core
- Better than current integrated graphics processors on laptops
- Should be commercially available starting 2011

CPU/GPU Hybrid (cont'd)

- Give this new technology some time
  - Hardware is always faster than software
  - For ex: the 64-bit transition has been very slow and gradual
- Powerful enough for high-end uses?
  - Separate GPUs evolve rapidly
  - Integrated GPUs for extended periods
  - Hybrid architectures won't be able to keep up

Getting Started with CUDA

- What do you need?
  - CUDA-enabled GPU (list here)
  - Driver for your device
  - CUDA software (free here)
  - Microsoft Visual Studio / Supported Linux version
- Install the CUDA software:
  - CUDA Toolkit: tools to build and compile
  - GPU Computing SDK: sample projects and source codes
- Test your installation with the samples
- For tips: CUDA Programming Guide

Getting Started with Jacket

- What is Jacket?
  - Developed by AccelerEyes
  - Accelerates MatLab code on CUDA-enabled GPUs
  - Family of products:
    - Jacket SDK (Development kit
    - Jacket MGL (Multi-GPU support)
    - Jacket Graphics Library
- What do you need?
  - CUDA-enabled GPU
  - Driver for your device
  - Matlab license (R2006B or greater)

Getting Started with Jacket (cont'd)

- How does it work?
  - Tag your data as GPU-data
    - A = rand(n)
    - A = gsingle(rand(A)) or A = grand(n)
  - Jacket-specific functions
    - Declaring inputs and outputs
    - Allocating memory to them
    - Calling the CUDA kernel
  - Graphics Library
    - Improves visualizations
    - Included with all Jacket licenses
    - Specific GPU function: gplot, gsurf,...
- Documentation is available here

What to remember

- GPGPU = General-Purpose computing on Graphics Processing Units
  - Why: increasing demand for computational power
  - Example: physics simulations
- CUDA = Compute Unified Device Architecture
  - Specific terminology : kernel, grid, blocks...
  - Different programming concepts
  - How to make a simple kernel
  - Optimizing this kernel

What to remember

- GPGPU has a bright future
  - Enables you to improve your existing code
  - New architectures are arriving (CPU/GPU Fusion)
- To get started, check out:
  - The CUDA toolkits from nVidia (free!)
  - OpenCL (free!)
  - Jacket for MatLab from Accelereyes (only 999$..)

References

- John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware
- Jared Hoberock and David Tarjan, Stanford University: Lecture 1: Introduction to Massively Parallel Computing
- Jared Hoberock and David Tarjan, Stanford University: Lecture 2: GPU History and CUDA Programming Basics

References (cont'd)

- nVidia. CUDA C Getting Started Guide For Microsoft Windows
- nVidia. CUDA C Programming Guide
- nVidia. CUDA Technical Training Volume I: Introduction to CUDA Programming
- nVidia: CUDA Technical Training Volume II: CUDA Case Studies

References (cont'd)

Further reading (most of the references can be found here):

- nVidia. GPU Computing Developer Home Page. `http://developer.nvidia.com/object/gpucomputing.html`