UPPSALA UNIVERSITET

GPU/CUDA programming

Dimitar Lukarski

Division of Scientific Computing
Department of Information Technology
Uppsala Programming for Multicore Architectures Research Center (UPMARC)
Uppsala University

Programming of Parallel Computers, March, 2014

UPPSALA UNIVERSITET

What You Should Know...

- Programming in C/C++
- Basics in hardware - CPU/Accelerators
- Concepts of parallel programming
- To have practice - OpenMP, pthreads,...
- Know some metric - Amdahl's law, Gustafson's law,...
- Parallel programming is FUN!

Outline

UPPSALA UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

Why GP-GPU?

Features

- High performance computing
- High bandwidth
- High parallelism

Architecture

- The largest part of the chip is dedicated for computing
- Smaller section is for local memories
- None (almost) for control logic (ILP, coherence protocols, branching, ...)

Price!

- Low - after all we need to buy this device
- The main market is the game/graphic industry

D. Lukarski, March, 2014, Uppsala

GP-GPU

GP-GPU = General Purpose - Graphical Processing Unit

- It becomes a factor in the HPC world
- Main target (numbers) is the Personal Super Computer

Comparison to UPPMAX

- Tintin: 160 × dual Opteron 6220
- Each Opteron 6220 = 192 GFlop/s
- In total (theoretical) = 61.4 TFlop/s

Equivalent GPU-based configuration

- K20 device = 1.17 TFlop/s
- 53 cards = 62 TFlop/s
- 14 nodes with 4 GPUs (56 cards)

UPPSALA UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

GP-GPU

GP-GPU as

- main compute power
- additional compute power
- *depends on the algorithm*

Main feature of the GPU – **Scalable Architecture!**

- Designed to be scalable
- Can perform many many many parallel threads

D. Lukarski, March, 2014, Uppsala

UPPSALA UNIVERSITET

GPU History

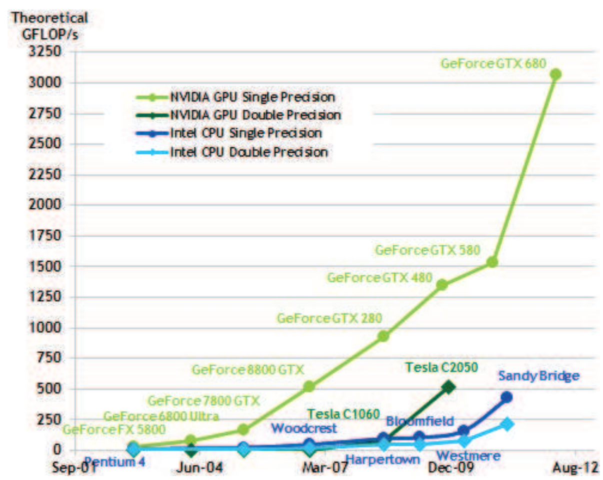


Figure 1 Floating-Point Operations per Second for the CPU and GPU

UPPSALA UNIVERSITET

GPU History

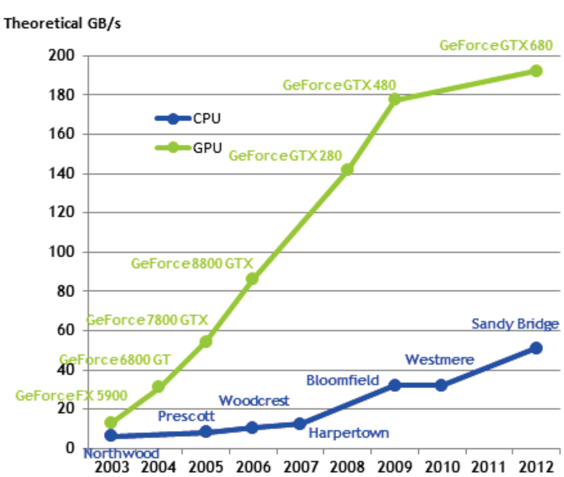


Figure 2 Memory Bandwidth for the CPU and GPU

Expectable Speed-ups

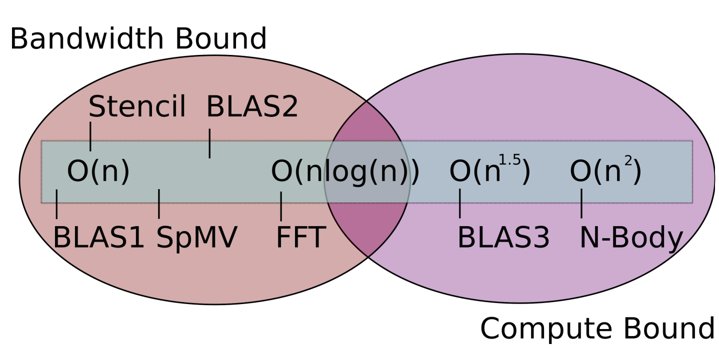
It depends on the

- Complexity
- Data dependency
- Branching
- Locality
- Your programming skills

Pure numbers

- Compute performance 4x-10x faster than CPU
- Bandwidth 4x-12x faster than CPU

Expectable Speed-ups



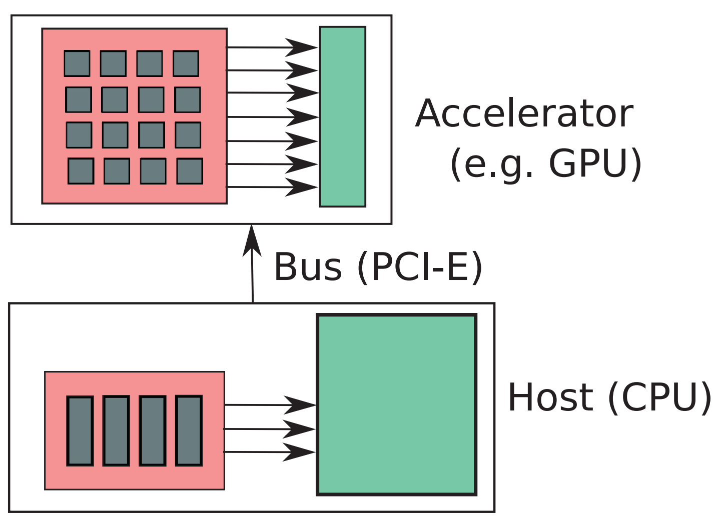
Expectable Speed-ups

Typicall numbers - 0.1x – 100x

- Monte Carlo Methods
- Molecular Dynamics
- Statistics
- Image analysis
- Dense Linear Algebra
- Sparse Linear Algebra

Note that too high/low numbers are typically attributed to bad CPU/GPU implementation!

GPU in a Computer



CPU - GPU

CPU:

- A few (big/fat) cores
- Big caches (MB)
- General purpose (HDD, Network, Sound ,...)
- Parallelism = none (seq) to 2× # cores

GPU:

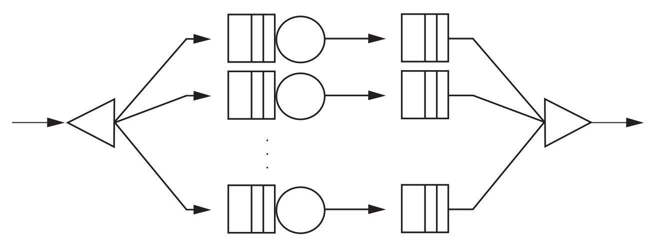
- Many small cores
- Small caches (KB)
- Optimized for streaming data
- Parallelism = > hundred(s)

UPPSALA UNIVERSITET

Overview - fork-join model

You should know

- OpenMP/pthreads/MPI
- Fork-join model



UPPSALA UNIVERSITET

Overview - fork-join model

This allows you to have

- long and complex threads
- locking mechanism / global barriers
  - for all threads
- global message passing
  - for all threads

Well, the GPU cannot do that!

UPPSALA UNIVERSITET

Example

$A + B = C,\ A, B, C \in \mathbb{R}^{N,N}$

- No dependency between the elements

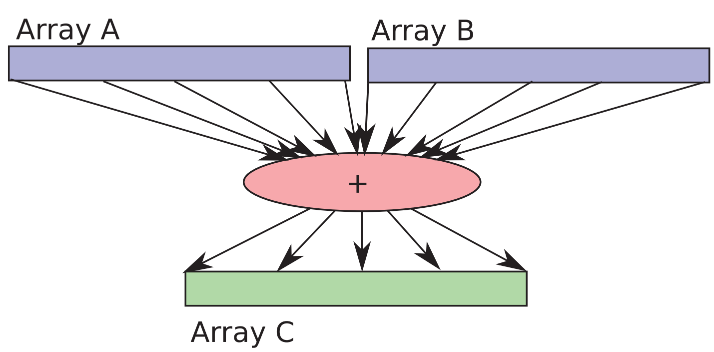
```
#pragma omp parallel for
for (int i=0; i<N; ++i)
for (int j=0; j<N; ++j)
c[i][j] = a[i][j] + b[i][j];
```

- Fork-join model
- Similar with pthreads/MPI

UPPSALA UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

Streaming Model



- Kernel +
- Implicit indexing

Pure Streaming Model

- No data dependency!
  - No global barriers
  - No global communication
- Very high level of parallelism
  - In our $A + B = C$ example
  - For $N = 1000$, parallelism $= 1M$
- The origins of this model come from the graphics
  - Every pixel is independent of the other
  - Rendering / Ray-tracing algorithms

UPPSALA
UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

D. Lukarski, March, 2014, Uppsala

UPPSALA UNIVERSITET

General Overview

CUDA is

- Language extension (C,C++,Fortran)
- Based on stream/data programming model

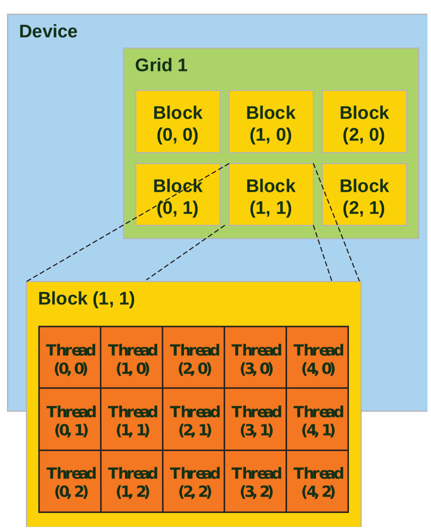
You need to

- Write a **kernel**
- Perform operations wrt the kernel **IDs**

UPPSALA UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

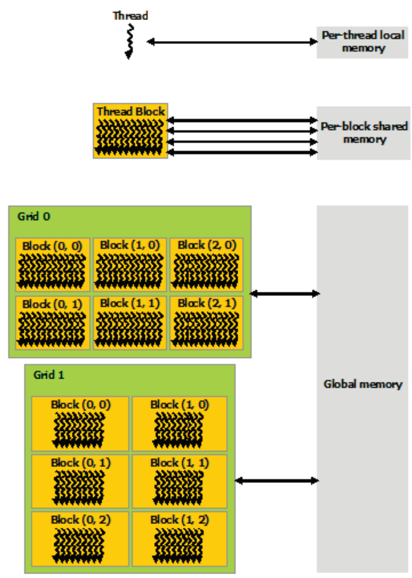
Grid/Thread Blocks/Threads



D. Lukarski, March, 2014, Uppsala

UPPSALA UNIVERSITET

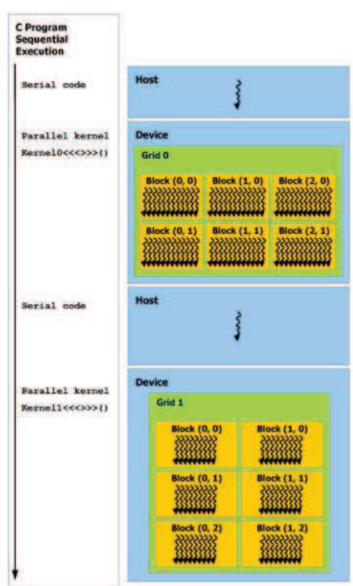
Grid/Thread Blocks/Threads



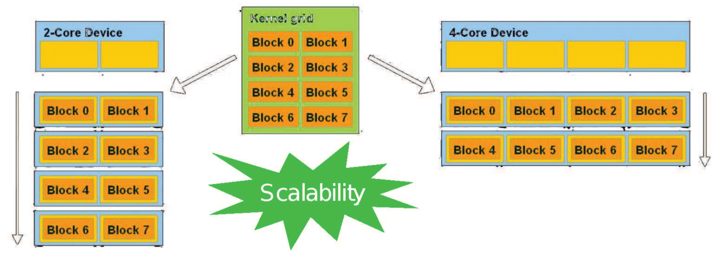
UPPSALA UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

Grid/Thread Blocks/Threads



CUDA Model is Scalable!



Example: Vector Update

$x = x + \alpha y$, where $x, y \in \mathbb{R}^N$ and $\alpha \in \mathbb{R}$

CPU code:

```
for (int i=0; i<N; ++i)
x[i] = x[i] + alpha*y[i];
```

GPU code:

```
__global__ void
vupdate(const int n, const double *y,
        const double alpha, const double *x) {
int i = blockIdx.x*blockDim.x + threadIdx.x;
if (i < n)
 x[i] = x[i] + alpha*y[i];
}
// ...
int nblocks = n/256 + 1;
vupdate<<<nblocks, 256>>>(n, y, alpha, x);
```

Kernel

CUDA Kernel:

- All threads run the **same** kernel
- The different path is based on the kernel **ID**

Kernel IDs are

- **thredIDx** - Thread ID
- **blockIDx** - Block ID
- **blockDim** - Block dimension
- **gridDim** - Grid dimension

Function declaration

- __**global**__ - GPU code / called from host
- __**device**__ - GPU code / called from GPU
- __**host**__ - host code / called from host

Kernel: Example 1/4

```
__global__ void kernel( int *a )
{
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] = 7;
}
```

Output:
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

Kernel: Example 2/4

```
__global__ void kernel( int *a )
{
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] = blockIdx.x*blockDim.x + threadIdx.x;
}
```

Output:
1 2 3 4 5 6 7 8 9 10 11 12 13 14

UPPSALA
UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

Kernel: Example 3/4

```
__global__ void kernel( int *a )
{
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] = blockIdx.x;
}
```

Output:
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
4 threads per block

D. Lukarski, March, 2014, Uppsala

Kernel: Example 4/4

```
__global__ void kernel( int *a )
{
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] = threadIdx.x;
}
```

Output:
0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
4 threads per block

UPPSALA UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

Memory Model

Thread:

- Registers
- Local memory (not really local)

Thread block:

- Shared memory
- Low latency - a few cycles
- High bandwidth

Grid (all threads):

- Device memory
- Latency 400-600 cycles
- Bandwidth 200 GB/s (K20)

D. Lukarski, March, 2014, Uppsala

UPPSALA
UNIVERSITET

Memory Model

Registers

- Where – on chip
- Access – R/W
- Scope – thread
- Lifetime – thread

Local Memory

- Where – off chip
- Access – R/W
- Scope – thread
- Lifetime – thread

UPPSALA
UNIVERSITET

Memory Model

Shared Memory

- Where – on chip
- Access – R/W
- Scope – block
- Lifetime – block

Global Memory

- Where – off chip
- Access – R/W
- Scope – All (dev+host)
- Lifetime – application

Memory Model

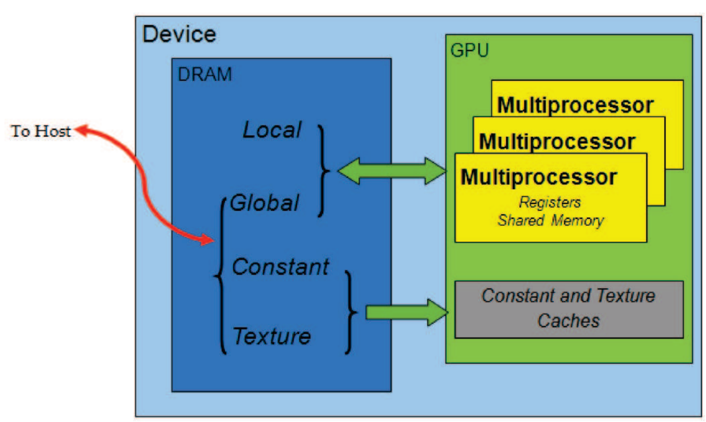
Contant Memory

- Where – off chip
- Access – R
- Scope – All(dev+host)
- Lifetime – application
- Cached!

Texture Memory

- Where – off chip
- Access – R
- Scope – All(dev+host)
- Lifetime – application
- Cached!

Memory Model



Memory Allocation

```
int n = 1024*sizeof(int);
int* d_a = NULL;

cudaMalloc((void**) &d_a, n);

// ...

cudaMemset(d_a, 0, n);

cudaFree(d_a);
```


UPPSALA
UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

Memory Moving

```
void main() {
int n = 256, num_bytes = n*sizeof(int);
int *d_a = NULL, *h_a = NULL;

h_a = (int*) malloc(num_bytes);

cudaMalloc( (void**) &d_a, num_bytes);
cudaMemset(d_a, 0, num_bytes);

cudaMemcpy(h_a, d_a, num_bytes,
           cudaMemcpyDeviceToHost);

free(h_a);
cudaFree(d_a);
}
```

D. Lukarski, March, 2014, Uppsala

UPPSALA UNIVERSITET

Memory Moving

cudaMemcpy(h_a,d_a,num_bytes, how); how:

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyHostToDevice

cudaMemcpyAsync()

- Asynchronous coping
- Return on the host immediately

UPPSALA UNIVERSITET

Launch a Kernel

- Define as __**global**__
- Syntax <<< >>>
- Specify number of threads/number of blocks

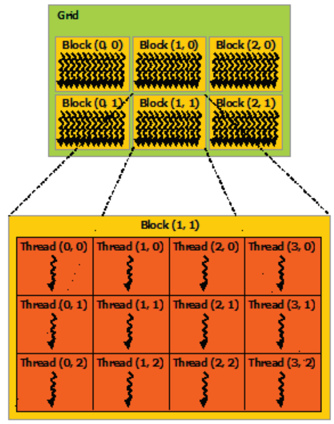
Kernel

```
__global__ void
vupdate(const int n, const double *y,
const double alpha, const double *x) {
int i = blockIdx.x*blockDim.x + threadIdx.x;
if (i < n)
x[i] = x[i] + alpha*y[i];
}
```

Calling

```
int nblocks = n/256 + 1;
vupdate<<<nblocks, 256>>>(n, y, alpha, x);
```

Grid/Thread Blocks/Threads



UPPSALA UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

Kernel

Kernels:

- The control is immediately return to the host
- No variable number of arguments
- No dynamic declarations (only at compiler time)
- Recursion is in CUDA 5

Blocks:

- The blocks must be independent
- The blocks are executed arbitrary
- No synchronization within the blocks
- Independence = Scalability

D. Lukarski, March, 2014, Uppsala

Compiler

Compilation:

- Add path to the bin/library (32/64)
- nvcc <program.cu>

Options

- Architecture (e.g. 2.0) -arch sm_20
- Linking -lcudart, -lcublas, -lcusparse

General Overview

GPUs have

- More core
- Less control units
- Less cache/local memories

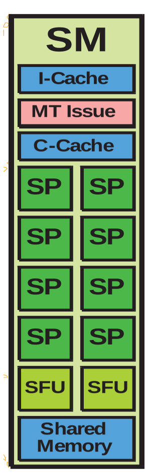
Scalar Processor

UPPSALA UNIVERSITET

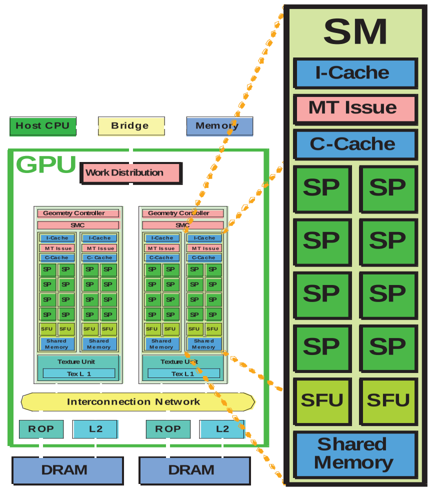
8x SP

UPPSALA UNIVERSITET

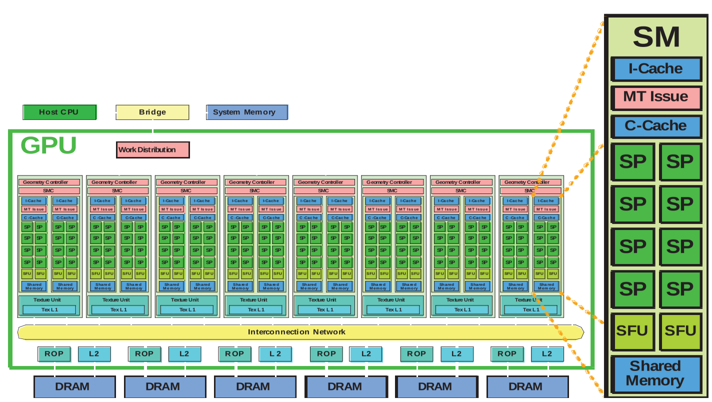
Stream Multiprocessor



32x SP

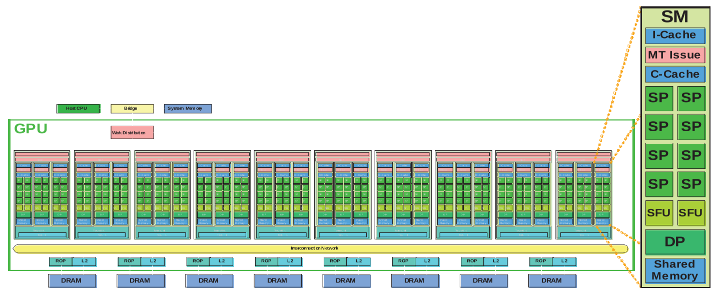


128x SP



UPPSALA UNIVERSITET

240x SP



UPPSALA UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

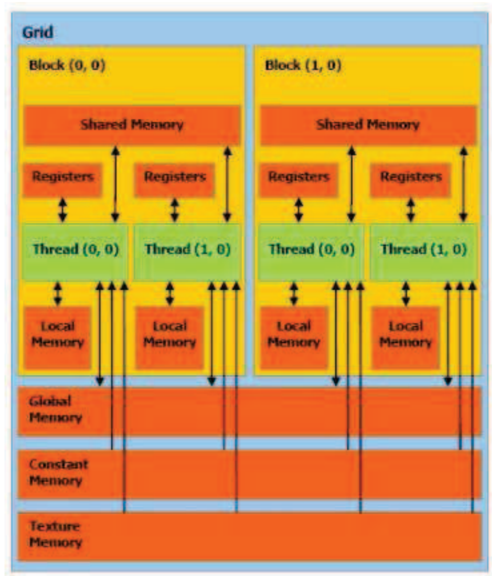
Kepler GPU has

2496 cuda cores!

CUDA

- Threads are executed on SP
- Thread Blocks are executed on SM (physically)
- 32 Threads = 1 Wrap
- 16 Threads = 1/2 Wrap execute a single memory access
- Blocks cannot migrate
- Several threads block can be active
- One kernel consists of a grid of blocks
- One/Multiple kernel can be performed at the same time

CUDA Memory



Shared Memory

- Very low latency
- Very high bandwidth
- 16-48KB per multi-processor
- Shared between 1-8 thread blocks
- Programmer is responsible for data movements (dataracing)
- Possible performance decrease due to bank-conflicts
- The size of the buffer should be determine at compiler time

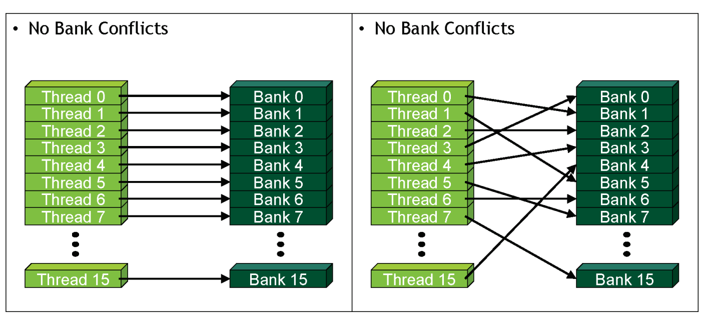
```
__shared__ int sm_buffer[SIZE];
```

Local Barrier

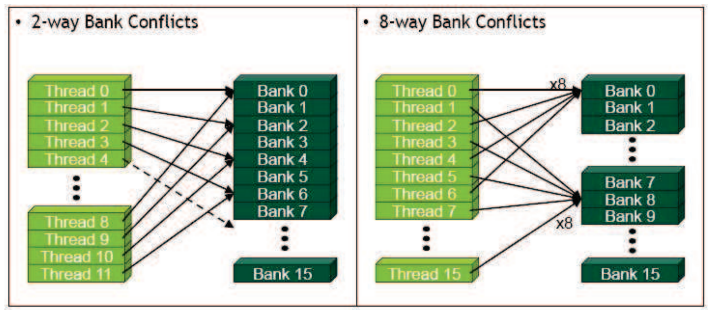
- Threads in a block can synchronize
- Thread Blocks cannot synchronize
- Typical usage – to avoid dataracing when using shared memory

```
// Declare shared memory
__shared__ int sm_buf[SIZE];

// Load data into sm_buff
Load(buff, sm_buff);

// Ensure all data is copied
__syncthreads();

// Compute data from sm_buff
Compute(sm_buff);
```


Bank Conflicts



Bank Conflicts



This example is for devices with lower compute capability

UPPSALA UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

Data Transfers

Host-GPU-Host

- Use pinned (non-pageable) memory
- Faster (2x) copy over PCI-E
- Do not use malloc/new and free/delete
- Use cudaHostAlloc/cudaFreeHost

GPU-GPU

- Fermi and Kepler
- Direct transfer with host interaction
- Requires new/special bus controller

D. Lukarski, March, 2014, Uppsala

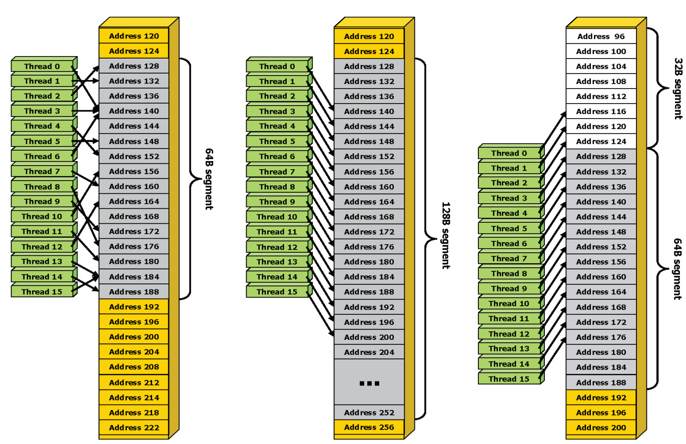
Memory Access/Alignment

- Global memory access
- GPU (as CPU) reads memory in segments
- 32B, 64B, 128B segments
- First address = multiple of the segment size
- This is called coalescing memory access

Memory access (kernel)

- Memory access is performed in half-wrap
- Goal: to have smallest number of transactions

Memory Access



UPPSALA
UNIVERSITET

Branching

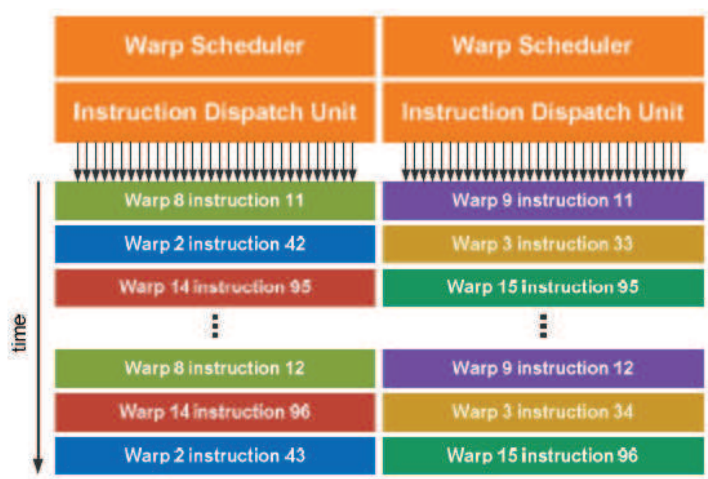
- One warp is physically executed
- i.e. 32 threads work at the same time
- But threads can take different path
- **if-else**
- This leads to unbalance in the SM

- Avoid **if (threadIdx.x % 2)** {...} **else** {...}
- To instead **if (threadIdx.x / WARP_SIZE % 2)** {...} **else** {...}

UPPSALA
UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

Warp Execution



D. Lukarski, March, 2014, Uppsala

Hiding Latency

- Switching different warps
- The number of warps depends of the kernel
- High register usage in the kernel = low number of active warps

Tips

- Smaller kernels − > more active warps − > more switching between warps− > better performance
- You can check/calculate this occupancy

UPPSALA
UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

Atomics

- There is no global synchronization
- But there are atomic operations
- **Add, Sub, Exch, Min, Max, Inc, Dec, CAS**
- **And, Or, Xor**

Example

- Histograms

D. Lukarski, March, 2014, Uppsala

Tools

CUDA-GDB

- Debugger
- Compile with **-G -g**
- cuda-gdb <program>
- Like GNU debugger

MEM-CHECK

- Memory checker
- cuda-memcheck <program>

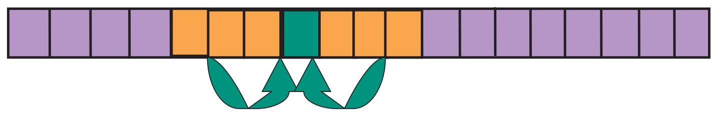
UPPSALA
UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

D. Lukarski, March, 2014, Uppsala

1D Stencil

- Array A, N elements
- Sum the neighboring elements with radius 3
- N inputs, N-3 outputs $= 2N$



Naive implementation

- For all i (1...N), load i-3,i-2,i+1,i,i+1,i+2,i+3
- 7 Loads + 1 Store $= 8N$

UPPSALA UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

1D Stencil

```
__global__ void stencil(const int* input,
                        int* output) {

int ix = blockIdx.x*blockDim.x + threadIdx.x;

int value = 0;

for (int i=0; i<=RADIUS, i++)
 value += input[ix+i];

for (int i=1; i<=RADIUS, i++)
 value += input[ix-i];

output[ix] = value;
}
```

D. Lukarski, March, 2014, Uppsala

UPPSALA UNIVERSITET

1D Stencil

Use shared memory

- Each thread block **blockDim.x** elements
- Each block loads **blockDim.x** $+ 2 \times 3$
- We call this -3 and +3 elements ghost cells
- Synchronize all threads in the block
- Compute the output results
- Write the results to the memory

UPPSALA UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

1D Stencil

```
__global__ void stencil(const int *input,
                        int *output){

__shared__ int s_a[SIZE+2*RADIUS];

int global_ix = blockIdx.x*blockDim.x
                + threadIdx.x;
int local_ix = threadIdx.x + RADIUS;

s_a[local_ix] = input[global_ix];

if ( threadIdx.x < RADIUS ){
 s_a[local_ix - RADIUS] =
              input[global_ix - RADIUS];
 s_a[local_ix + SIZE + RADIUS] =
 input[global_ix + SIZE + RADIUS];
}
```

UPPSALA UNIVERSITET

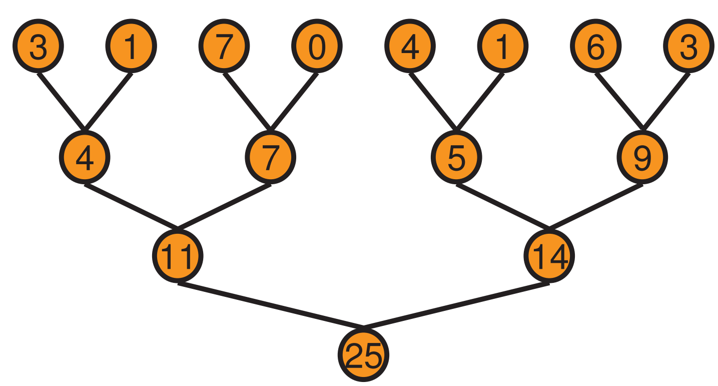
1D Stencil

```
__syncthreads();

int value = 0;

for( offset = -RADIUS; offset<=RADIUS; offset++)
 value += s_a[ local_ix + offset ];

output[global_ix] = value;
}
```

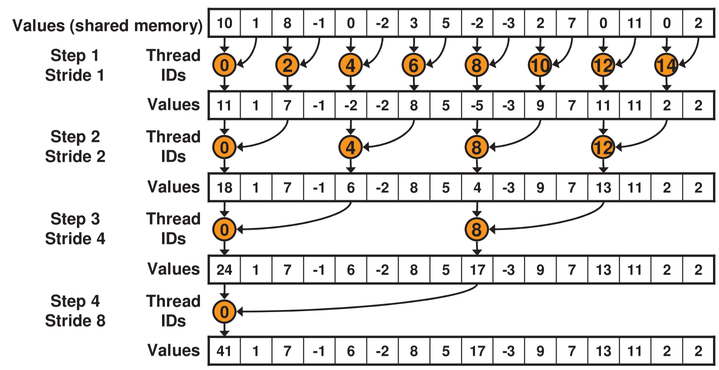

UPPSALA UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

Parallel Reduction



Parallel Reduction

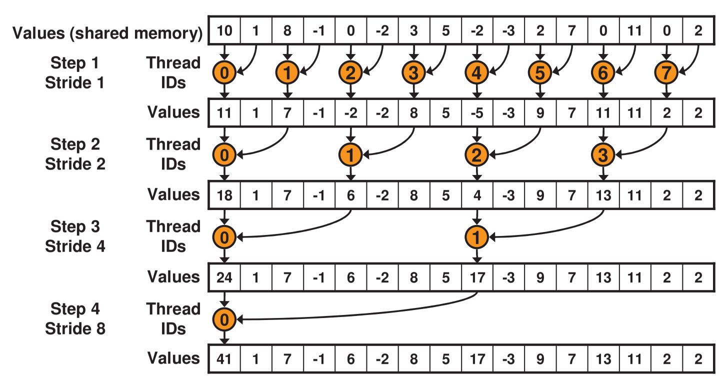


Warps diverge

UPPSALA
UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

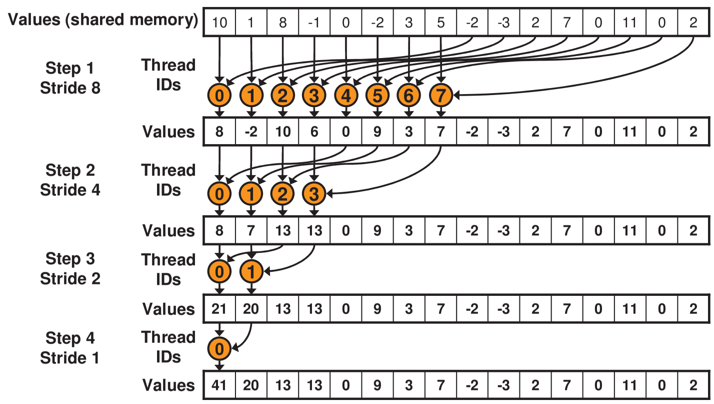
Parallel Reduction



Bank conflicts

UPPSALA UNIVERSITET

Parallel Reduction



UPPSALA UNIVERSITET

Parallel Reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative 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	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Note: very old hardware

UPPSALA
UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

Libraries

CUBLAS

- Dense operations

CUSPARSE

- Sparse operations
- Various formats

Thrust

- Parallel algorithms
- Similar to the C++ Standard Template Library (STL)

D. Lukarski, March, 2014, Uppsala

UPPSALA
UNIVERSITET

UPPSALA
UNIVERSITET

Outline
Intro and motivation
Streaming model
CUDA
NVIDIA GPUs
Optimization
Examples
Libraries
GPUs and UU

GPUs and UU

We do GPU computing and we interested in

- New algorithms
- Adapting old algorithms
- Developing software

Uppsala University is also

References (from this talk)

- CUDA 5.0 documentation
- NVIDIA/CUDA slides (google)