CUDA

Karl Ljungkvist

25 February 2016

GPU computing history

Early 2000's:

- GPUs becomming powerful
- Useful for non-graphics stuff?

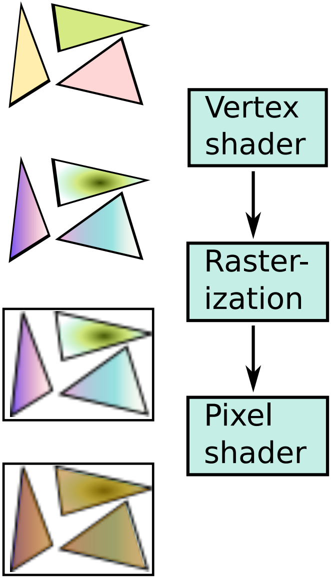
GPU computing history

Early 2000's:

- GPUs becomming powerful
- Useful for non-graphics stuff?

Shader programming:

- Exploit rendering pipeline
- Vertex and pixel shaders
- Vectors (colors, positions) and matrices natively
- Really messy!



GPU computing history

Early 2000's:

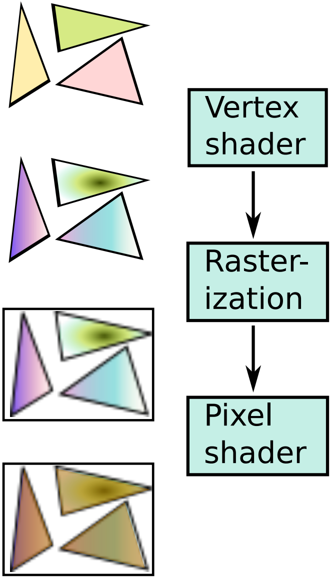
- GPUs becomming powerful
- Useful for non-graphics stuff?

Shader programming:

- Exploit rendering pipeline
- Vertex and pixel shaders
- Vectors (colors, positions) and matrices natively
- Really messy!

2006:

- Nvidia CUDA



CUDA Intro

Compute Unified Device Architecture

- Dedicated framework for GPU programming
- Programming language (C/C++ extension)
- Hardware and thread model
- Development toolkit

CUDA Intro

Compute Unified Device Architecture

- Dedicated framework for GPU programming
- Programming language (C/C++ extension)
- Hardware and thread model
- Development toolkit

Pros/Cons

++ Low-level (high performance)

CUDA Intro

Compute Unified Device Architecture

- Dedicated framework for GPU programming
- Programming language (C/C++ extension)
- Hardware and thread model
- Development toolkit

Pros/Cons

++ Low-level (high performance)

\- Low-level (hard to program)

CUDA Intro

Compute Unified Device Architecture

- Dedicated framework for GPU programming
- Programming language (C/C++ extension)
- Hardware and thread model
- Development toolkit

Pros/Cons

++ Low-level (high performance)

\- Low-level (hard to program)

\+ Mature (debugger, profiler, ...)

CUDA Intro

Compute Unified Device Architecture

- Dedicated framework for GPU programming
- Programming language (C/C++ extension)
- Hardware and thread model
- Development toolkit

Pros/Cons

++ Low-level (high performance)

\- Low-level (hard to program)

\+ Mature (debugger, profiler, ...)

\- Nvidia only

CUDA Intro

Compute Unified Device Architecture

- Dedicated framework for GPU programming
- Programming language (C/C++ extension)
- Hardware and thread model
- Development toolkit

Pros/Cons

++ Low-level (high performance)
- Low-level (hard to program)
+ Mature (debugger, profiler, ...)
- Nvidia only
+ Portable across Nvidia GPUs

OpenCL & CUDA

Also: OpenCL

- Open standard
- Cross platform (GPUs, multicores, FPGA, etc)
- Low-level (high performance)

OpenCL & CUDA

Also: OpenCL

- Open standard
- Cross platform (GPUs, multicores, FPGA, etc)
- Low-level (high performance)

Why CUDA?

- OpenCL must be tuned to perform
- CUDA performs better
- Not as mature
- Messy to program

OpenCL & CUDA

Also: OpenCL

- Open standard
- Cross platform (GPUs, multicores, FPGA, etc)
- Low-level (high performance)

Why CUDA?

- OpenCL must be tuned to perform
- CUDA performs better
- Not as mature
- Messy to program

They are similar:

- Very similar programming models
- Can easily convert $\sim$ CUDA $\Leftrightarrow$ OpenCL

Code example

Vector addition:

$$\mathbf{x} := \mathbf{x} + \mathbf{y}$$

CPU function:

```
void vec_add(float *x, const float *y, int N) {
  for(int i=0; i<N; ++i)
    x[i] = x[i] + y[i];
}
```

Code example

Vector addition:

$$\mathbf{x} := \mathbf{x} + \mathbf{y}$$

CPU function:

```
void vec_add(float *x, const float *y, int N) {
  for(int i=0; i<N; ++i)
    x[i] = x[i] + y[i];
}
```

CUDA *kernel*:

```
__global__ void vec_add(float *x, const float *y, int N) {
  int i = threadIdx.x;
  x[i] = x[i] + y[i];
}
```

Code example

Vector addition:

$$\mathbf{x} := \mathbf{x} + \mathbf{y}$$

CPU function:

```
void vec_add(float *x, const float *y, int N) {
  for(int i=0; i<N; ++i)
    x[i] = x[i] + y[i];
}
```

CUDA *kernel*:

```
__global__ void vec_add(float *x, const float *y, int N) {
  int i = threadIdx.x;
  x[i] = x[i] + y[i];
}
```

Call:

```
vec_add<<<1,N>>>(x,y,N);
```

Question: What problem with shader programming was solved with the release of CUDA?

- Difficult to program
- Code was not portable
- Performance was bad

Question: What problem with shader programming was solved with the release of CUDA?

- Difficult to program
- Code was not portable
- Performance was bad

Question: What is the main drawback with CUDA?

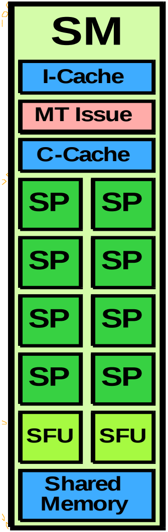
- Hard to profile and debug
- Code is not portable
- Programming language is non-intuitive

Hardware Model

Hardware model

Fundamental entity:

- *CUDA core* or *Streaming Processor* (*SP*)



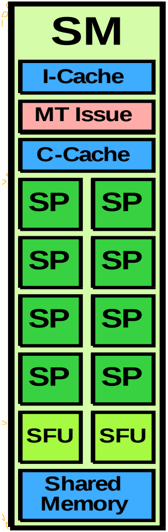
Hardware model

Fundamental entity:

- *CUDA core* or *Streaming Processor* (*SP*)

Streaming Multiprocessor (SM):

- A collection of CUDA cores (8 / 32 / 192)
- All cores in one SM run the same instructions
- Has some fast, shared cache memory
- Can synchronize



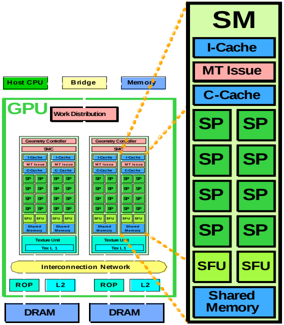
Hardware model

The CUDA device:

- A collection of SMs + memory

A scalable model:

- 2x2 8-core SMs, 32 cores



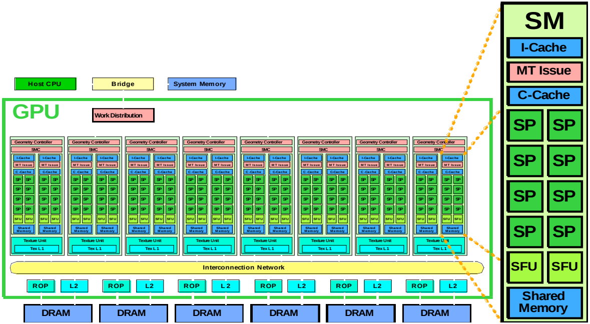
Hardware model

The CUDA device:

- A collection of SMs + memory

A scalable model:

- 8x2 8-core SMs, 128 cores



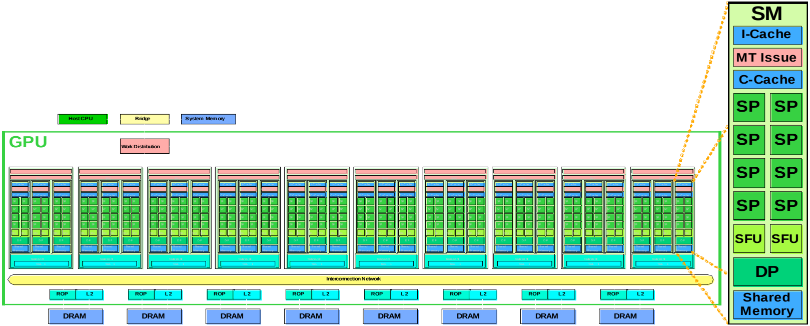
Hardware model

The CUDA device:

- A collection of SMs + memory

A scalable model:

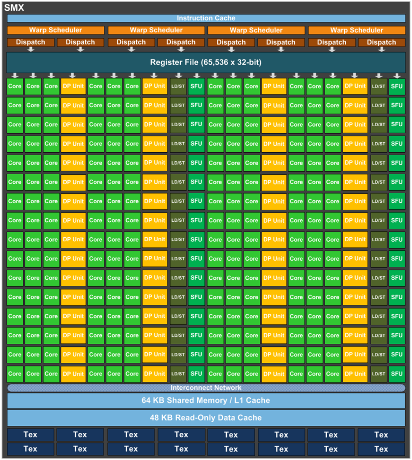
- 10x3 8-core SMs, 240 cores



Hardware model

Recent architecture:

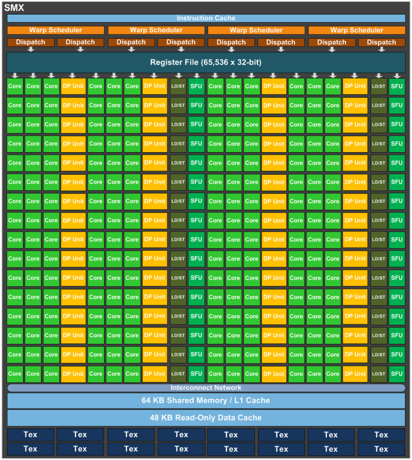
- Kepler architecture
- SMX: 192 simpler cores



Hardware model

Recent architecture:

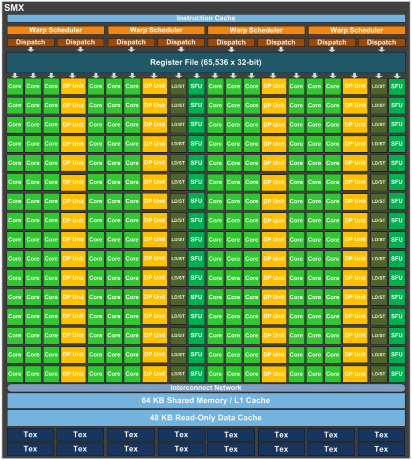
- Kepler architecture
- SMX: 192 simpler cores
- GPU GK110 (K20): 15xSMX



Hardware model

Recent architecture:

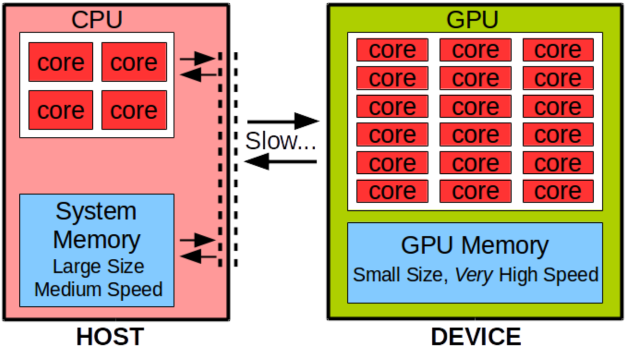
- Kepler architecture
- SMX: 192 simpler cores
- GPU GK110 (K20): 15xSMX
- 2880 cores!



Hardware model

System:

- The Device connected to a *Host* (CPU)



Question: Which are correct?

The CUDA cores in an SM...

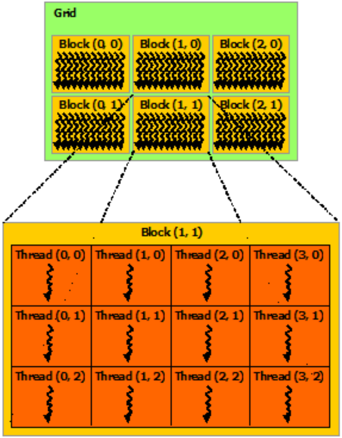
- Execute individual instructions
- Can synchronize
- Has a fast shared cache

Thread Model

Thread model

Thread hierarchy:

- *Threads* executing a *kernel*
- Threads grouped into *blocks*
- Threads in a block run together
- Blocks organized in a *grid*



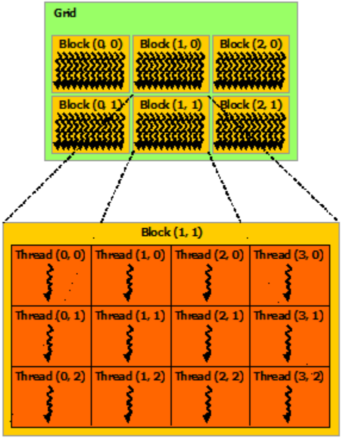
Thread model

Thread hierarchy:

- *Threads* executing a *kernel*
- Threads grouped into *blocks*
- Threads in a block run together
- Blocks organized in a *grid*

Block size:

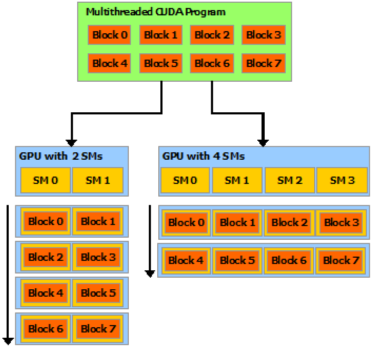
- Configurable
- Optimum depends on application and hardware
- Often, large blocks are better
- Typically, ∼ 256-1024 threads/block



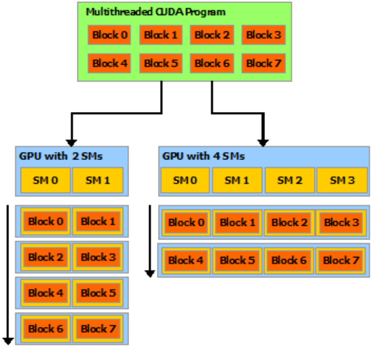
Thread model

At runtime:

- Blocks assigned to SMs
- Again, scalability



Thread model



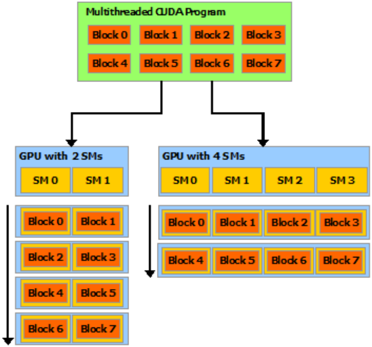
At runtime:

- Blocks assigned to SMs
- Again, scalability

Warps:

- Smaller subdivision of a block
- One *warp* = 32 threads
- Threads in warp executed in *SIMD* (Single Instruction Multiple Data) fashion

Thread model



At runtime:

- Blocks assigned to SMs
- Again, scalability

Warps:

- Smaller subdivision of a block
- One *warp* = 32 threads
- Threads in warp executed in *SIMD* (Single Instruction Multiple Data) fashion

The warps are what is scheduled:

- Want high number of active warps, or *occupancy*
- Usage of resources (registers, shared memory, etc) limits occupancy
- Not handled explicitly when programming, but can matter for performance

Thread model

Branch divergence:

- Threads in warp execute simultaneously on the SM
- $\Rightarrow$ must execute the same instruction
- Branch divergence can hurt performance

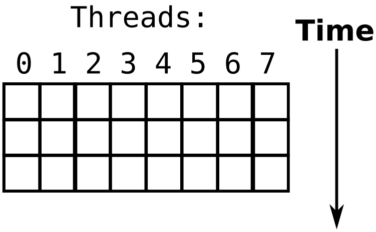
Thread model

Branch divergence:

- Threads in warp execute simultaneously on the SM
- $\Rightarrow$ must execute the same instruction
- Branch divergence can hurt performance

Example:

```
if(cond(thrid)):
  do A
else: do B
```



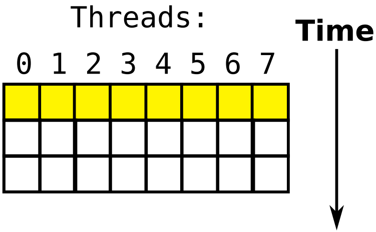
Thread model

Branch divergence:

- Threads in warp execute simultaneously on the SM
- $\Rightarrow$ must execute the same instruction
- Branch divergence can hurt performance

Example:

```
if(cond(thrid)):
  do A
else: do B
```



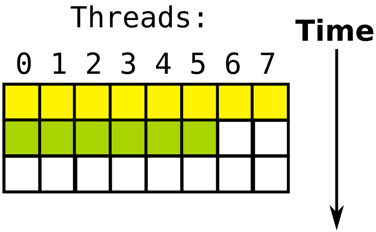
Thread model

Branch divergence:

- Threads in warp execute simultaneously on the SM
- $\Rightarrow$ must execute the same instruction
- Branch divergence can hurt performance

Example:

```
if(cond(thrid)):
  do A
else: do B
```



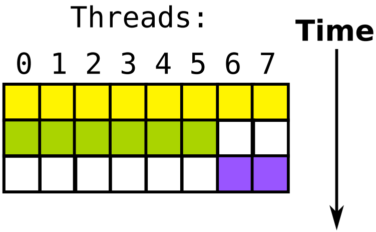
Thread model

Branch divergence:

- Threads in warp execute simultaneously on the SM
- $\Rightarrow$ must execute the same instruction
- Branch divergence can hurt performance

Example:

```
if(cond(thrid)):
  do A
else: do B
```



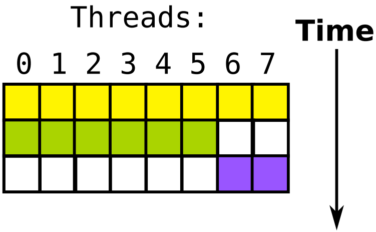
Thread model

Branch divergence:

- Threads in warp execute simultaneously on the SM
- $\Rightarrow$ must execute the same instruction
- Branch divergence can hurt performance

Example:

```
if(cond(thrid)):
  do A
else: do B
```



- *75% efficiency loss!*

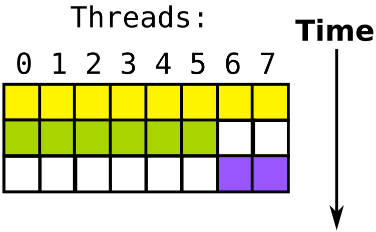
Thread model

Branch divergence:

- Threads in warp execute simultaneously on the SM
- ⇒ must execute the same instruction
- Branch divergence can hurt performance

Example:

```
if(cond(thrid)):
  do A
else: do B
```



- *75% efficiency loss!*

Note: Threads in the same block but different warps do not have this problem.

Question: Correct? All threads in a block execute on an SM simultaneously.

- Yes
- No

Question: Correct? All threads in a block execute on an SM simultaneously.

- Yes
- No

Question: Correct? All warps in a block execute on the same SM.

- Yes
- No

CUDA Kernels

CUDA Kernels

Types of functions:

`__global__` Device code called from host – *kernel*

`__device__` Device code called from device

`__host__` Host code called from host (usual functions)

CUDA Kernels

Types of functions:

`__global__`	Device code called from host – *kernel*
`__device__`	Device code called from device
`__host__`	Host code called from host (usual functions)

Identifying threads:

- Thread in block:
  `threadIdx`
- Block in grid:
  `blockIdx`
- Size of block:
  `blockDim`

CUDA Kernels

Types of functions:

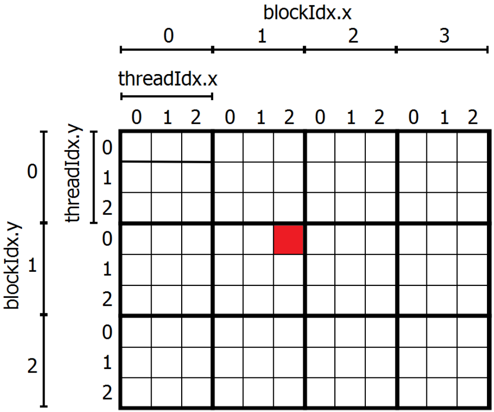
`__global__` Device code called from host – *kernel*

`__device__` Device code called from device

`__host__` Host code called from host (usual functions)

Identifying threads:

- Thread in block: `threadIdx`
- Block in grid: `blockIdx`
- Size of block: `blockDim`



dim3

- Type: `dim3` – 1D, 2D, or 3D

CUDA Kernels

Kernel invocation:

```
kernel<<<grid_dim, block_dim>>>(args);
```

Configuration parameters:

- `block_dim` – size of thread block (`dim3`)
- `grid_dim` – blocks per grid (`dim3`)

Example: matrix addition

Code:

```
/* kernel */
__global__ void matrix_sum(float *C, const float *A,
                           const float *B, int N) {
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  int j = threadIdx.y + blockIdx.y*blockDim.y;
  if(i<N && j<N)
    C[i*N+j] = A[i*N+j] + B[i*N+j];
}
```

Invocation:

```
...
  /* kernel configuration */
  dim3 block_dim(8,8);
  int num_blocks = 1+ (N-1)/8;
  dim3 grid_dim(num_blocks,num_blocks);
  /* kernel launch */
  matrix_sum<<<grid_dim,block_dim>>>(C,A,B,N);
...
```

Questions: Which are correct?

A kernel...

- is a function running on the device,
- is started from the host,
- always runs the same number of threads.

Questions: Which are correct?

A kernel...

- is a function running on the device,
- is started from the host,
- always runs the same number of threads.

The block dimension...

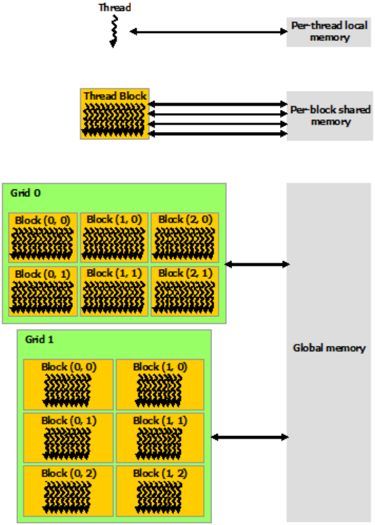
- shouldn't be too large for good performance,
- must be known at compile time,
- can be three-dimensional.

Memory in CUDA

Memory in CUDA

Types of memory:

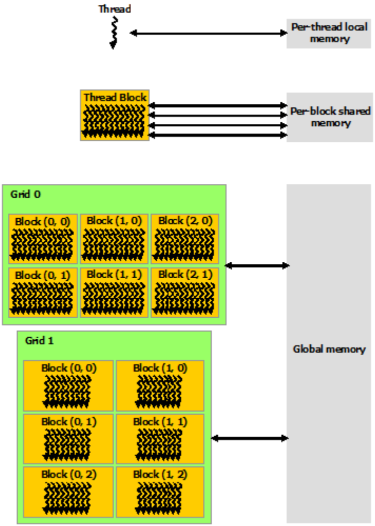
- Single thread has *registers*
  - $\leq$ 255 32-bit regs
  - 0 cycle



Memory in CUDA

Types of memory:

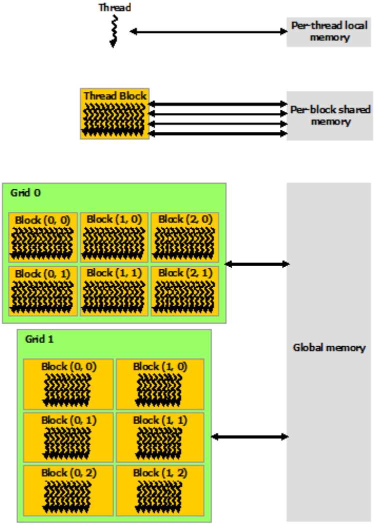
- Single thread has *registers*
  - $\leq$ 255 32-bit regs
  - 0 cycle
- Threads in block share memory – *shared* memory
  - 48 kB
  - $\sim$ 50 cycles
  - 1.5-2TB/s



Memory in CUDA

Types of memory:

- Single thread has *registers*
  - $\leq$ 255 32-bit regs
  - 0 cycle
- Threads in block share memory – *shared* memory
  - 48 kB
  - $\sim$ 50 cycles
  - 1.5-2TB/s
- Main device memory – *global* memory
  - $\sim$ 10 GB
  - $\sim$ 500 cycles
  - $\sim$ 200 GB/s



Memory in CUDA

Allocating device memory:

- `cudaMalloc` – allocates *global* memory on device
- `cudaFree` – frees it again
- `cudaMemset` – used for initializing memory

Used exactly like:

- `malloc`
- `free`
- `memset`

Memory in CUDA

Allocating device memory:

- `cudaMalloc` – allocates *global* memory on device
- `cudaFree` – frees it again
- `cudaMemset` – used for initializing memory

Used exactly like:

- `malloc`
- `free`
- `memset`

Device and host pointers:

- Pointers *either* valid on host or device

Memory in CUDA

Host-device transfer – `cudaMemcpy`:

- Host to device:

```
cudaMemcpy(x_dev,x_host,num_bytes,
           cudaMemcpyHostToDevice);
```

- Device to host:

```
cudaMemcpy(x_host,x_device,num_bytes,
           cudaMemcpyDeviceToHost);
```

Memory in CUDA

Host-device transfer – `cudaMemcpy`:

- Host to device:

```
cudaMemcpy(x_dev,x_host,num_bytes,
           cudaMemcpyHostToDevice);
```

- Device to host:

```
cudaMemcpy(x_host,x_device,num_bytes,
           cudaMemcpyDeviceToHost);
```

- PCIe $\sim$ 3-8 GB/s

Memory in CUDA

Host-device transfer – `cudaMemcpy`:

- Host to device:

```
cudaMemcpy(x_dev,x_host,num_bytes,
           cudaMemcpyHostToDevice);
```

- Device to host:

```
cudaMemcpy(x_host,x_device,num_bytes,
           cudaMemcpyDeviceToHost);
```

- PCIe $\sim$ 3-8 GB/s

Also `cudaMemcpyAsync`:

- Non-blocking communication
- Overlap communication and computation
- Cf. `MPI_Isend`

Memory in CUDA

Example:

```
void main() {
  int n = 256;
  int num_bytes = n*sizeof(int);
  int *x_dev, *x_host;
  /* allocate memory */
  x_host = (int*) malloc(num_bytes);
  cudaMalloc(&x_dev, num_bytes);
  /* set to 0 */
  cudaMemset(x_dev, 0, num_bytes);
  /* copy memory to host */
  cudaMemcpy(x_host, x_dev, num_bytes,
             cudaMemcpyDeviceToHost);
  /* free up memory */
  free(x_host);
  cudaFree(x_dev);
}
```

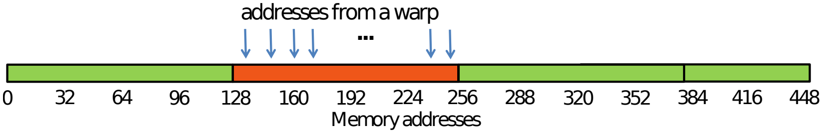
Efficient memory usage

Global memory access:

- Threads in warp access memory together
- Hardware minimizes number of transactions

Coalesced access 1:

- Threads read contiguous memory – single transaction



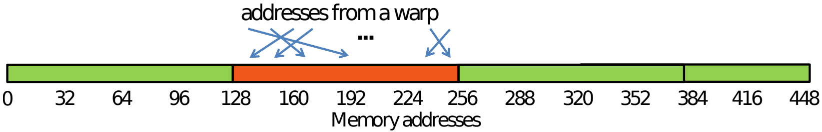
Efficient memory usage

Global memory access:

- Threads in warp access memory together
- Hardware minimizes number of transactions

Coalesced access 2:

- Contiguous but permuted access – single transaction



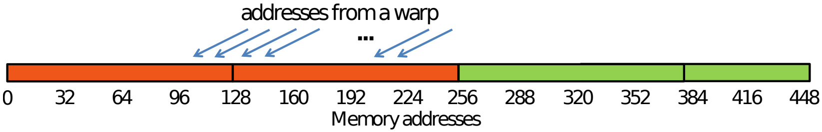
Efficient memory usage

Global memory access:

- Threads in warp access memory together
- Hardware minimizes number of transactions

Uncoalesced access 1:

- Contiguous but misaligned – two transactions



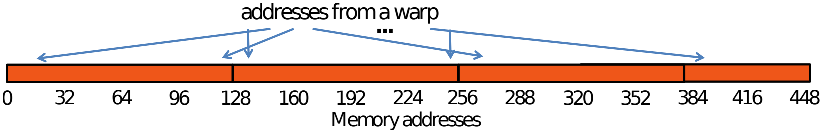
Efficient memory usage

Global memory access:

- Threads in warp access memory together
- Hardware minimizes number of transactions

Uncoalesced access 2:

- Non-contiguous access – *N* transactions



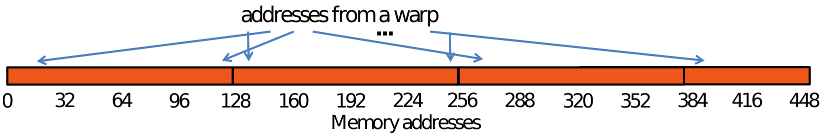
Efficient memory usage

Global memory access:

- Threads in warp access memory together
- Hardware minimizes number of transactions

Uncoalesced access 2:

- Non-contiguous access – *N* transactions



Note: Nowadays, caches help a bit

Efficient memory usage

Compare with CPU:

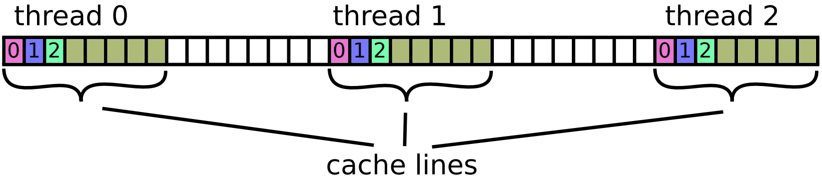
- Local caching for each thread
- Locality for each thread

Efficient memory usage

Compare with CPU:

- Local caching for each thread
- Locality for each thread

Example:

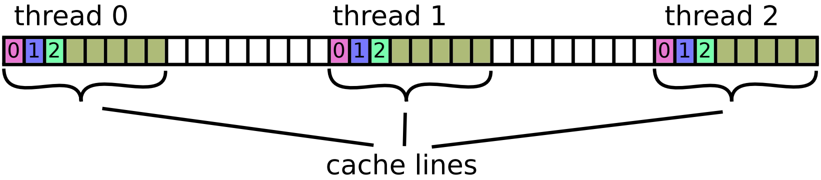


Efficient memory usage

Compare with CPU:

- Local caching for each thread
- Locality for each thread

Example:



Opposite access pattern!

Efficient memory usage

Array-of-Structure vs Structure-of-Array:

- Access collection of 3D points

Efficient memory usage

Array-of-Structure vs Structure-of-Array:

- Access collection of 3D points

```
/* Array of Structure */
typedef struct {
  float x,y,z;
} aos_t;
aos_t aos[1000];

/* access */
float xval = soa[i].x;
float yval = soa[i].y;
float zval = soa[i].z;
```

```
/* Structure of Array */
typedef struct {
  float x[1000];
  float y[1000];
  float z[1000];
} soa_t;
soa_t soa;

/* access */
float xval = soa.x[i];
float yval = soa.y[i];
float zval = soa.z[i];
```

Efficient memory usage

Array-of-Structure vs Structure-of-Array:

- Access collection of 3D points

```
/* Array of Structure */
typedef struct {
  float x,y,z;
} aos_t;
aos_t aos[1000];

/* access */
float xval = soa[i].x;
float yval = soa[i].y;
float zval = soa[i].z;
```

```
/* Structure of Array */
typedef struct {
  float x[1000];
  float y[1000];
  float z[1000];
} soa_t;
soa_t soa;

/* access */
float xval = soa.x[i];
float yval = soa.y[i];
float zval = soa.z[i];
```

Question:
Which will perform best on a GPU?

- Array-of-Structure
- Structure-of-Array
- Does not matter

Efficient memory usage

Array-of-Structure vs Structure-of-Array:

- Access collection of 3D points

```
/* Array of Structure */
typedef struct {
  float x,y,z;
} aos_t;
aos_t aos[1000];

/* access */
float xval = soa[i].x;
float yval = soa[i].y;
float zval = soa[i].z;
```

```
/* Structure of Array */
typedef struct {
  float x[1000];
  float y[1000];
  float z[1000];
} soa_t;
soa_t soa;

/* access */
float xval = soa.x[i];
float yval = soa.y[i];
float zval = soa.z[i];
```

Question:
Which will perform best on a GPU?

- Array-of-Structure
- Structure-of-Array
- Does not matter

...or on a CPU?

- Array-of-Structure
- Structure-of-Array
- Does not matter

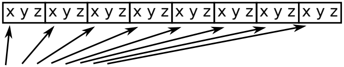
Efficient memory usage

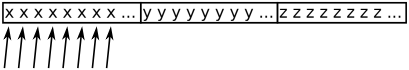
Array-of-Structure vs Structure-of-Array:

- Access collection of 3D points

```c
/* Array of Structure */
typedef struct {
  float x,y,z;
} aos_t;
aos_t aos[1000];

/* access */
float xval = soa[i].x;
float yval = soa[i].y;
float zval = soa[i].z;
```

```c
/* Structure of Array */
typedef struct {
  float x[1000];
  float y[1000];
  float z[1000];
} soa_t;
soa_t soa;

/* access */
float xval = soa.x[i];
float yval = soa.y[i];
float zval = soa.z[i];
```





Efficient memory usage

Shared memory

- Small and very fast memory shared by thread block
- E.g. user-managed cache, or scratchpad for cooperative algorithm
- Programmer responsible for avoiding race conditions

Usage:

```
__shared__ int buffer[SIZE];
```

Efficient memory usage

Shared memory

- Small and very fast memory shared by thread block
- E.g. user-managed cache, or scratchpad for cooperative algorithm
- Programmer responsible for avoiding race conditions

Usage:

```
__shared__ int buffer[SIZE];
```

Also:

- Automatic L1 cache
- Since 2010 (Fermi architecture)
- Configurable, 16kB+48kB / 48kB+16kB

Efficient memory usage

8-bank example:

Bank 7	7	15	23							
Bank 6	6	14	22							
Bank 5	5	13	21							
Bank 4	4	12	20							
Bank 3	3	11	19							
Bank 2	2	10	18							
Bank 1	1	9	17							
Bank 0	0	8	16							

Shared memory banks:

- 32 banks
- Consecutive 4B-*words* belong to different banks, cyclically

Efficient memory usage

Shared memory banks:

- 32 banks
- Consecutive 4B-*words* belong to different banks, cyclically

8-bank example:

Bank 7	7	15	23							
Bank 6	6	14	22							
Bank 5	5	13	21							
Bank 4	4	12	20							
Bank 3	3	11	19							
Bank 2	2	10	18							
Bank 1	1	9	17							
Bank 0	0	8	16							

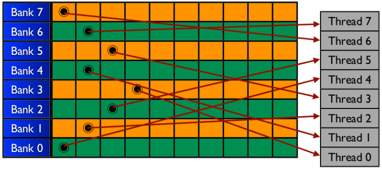
Bank conflicts:

- Penalty if threads in warp access same bank

Efficient memory usage

All different banks:

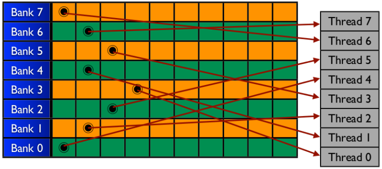
- No conflicts



Efficient memory usage

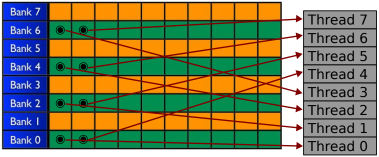
All different banks:

- No conflicts



2-way bank conflict:

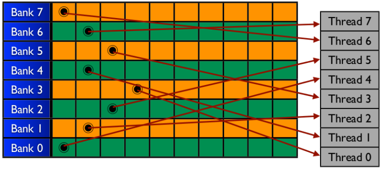
- Half performance



Efficient memory usage

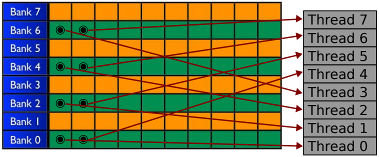
All different banks:

- No conflicts



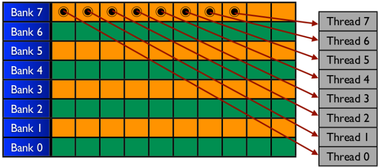
2-way bank conflict:

- Half performance



Full bank conflict:

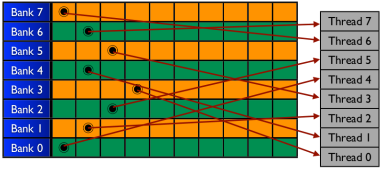
- 1/8 performance



Efficient memory usage

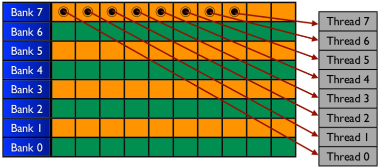
All different banks:

- No conflicts



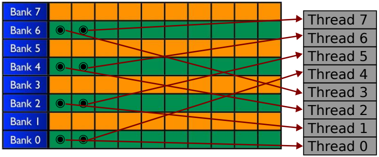
Full bank conflict:

- 1/8 performance



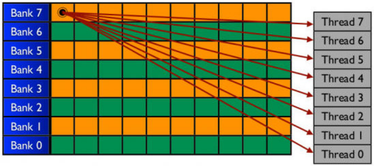
2-way bank conflict:

- Half performance



Broadcast:

- Threads access *same location* – no overhead



Question: Match the memory issues with the right memory type

- Uncoalesced access
- Bank conflicts
- Slow PCI-e bus

Synchronization

Synchronization in CUDA

Global synchronization:

- Does not exist!
- Implicit global synchronization at kernel boundaries

Synchronization in CUDA

Global synchronization:

- Does not exist!
- Implicit global synchronization at kernel boundaries

Synchronization within block:

- `__syncthreads()`
- Barrier for threads in block
- Avoid data race when using `shared` memory

Synchronization in CUDA

Global synchronization:

- Does not exist!
- Implicit global synchronization at kernel boundaries

Synchronization within block:

- `__syncthreads()`
- Barrier for threads in block
- Avoid data race when using `shared` memory

Example: array reversal

```
__global__ void reverse(int *d,
                        int n) {
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

Synchronization in CUDA

Global synchronization:

- Does not exist!
- Implicit global synchronization at kernel boundaries

Synchronization within block:

- `__syncthreads()`
- Barrier for threads in block
- Avoid data race when using `shared` memory

Example: array reversal

```
__global__ void reverse(int *d,
                        int n) {
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

Atomic operations:

- Perform one operation in thread-safe manner
- `atomicAdd`, `atomicMax`, etc

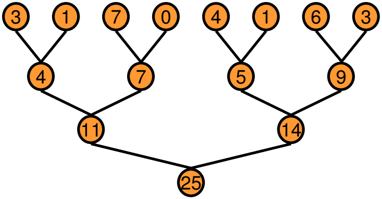
Example: Parallel Reduction

Important algorithm:

- Examples: Vector max, element sum/product, dot prod., ...
- Other algorithms solved with same technique

Vector sum

```
sum=0;
for(int i=0; i<N; ++i)
  sum += A[i];
```



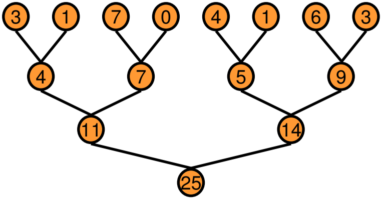
Example: Parallel Reduction

Important algorithm:

- Examples: Vector max, element sum/product, dot prod., ...
- Other algorithms solved with same technique

Vector sum

```
sum=0;
for(int i=0; i<N; ++i)
  sum += A[i];
```

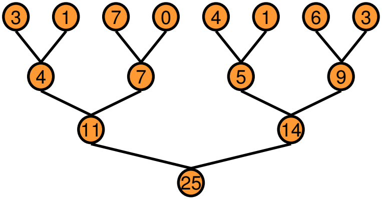


Also see:

- GPU lab
- Presentation: "Optimizing Parallel Reduction in CUDA", Mark Harris

  http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

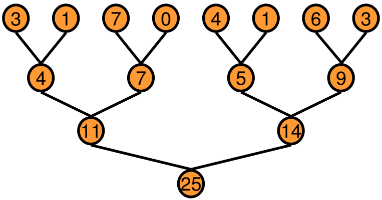
Example: Parallel Reduction



Vector sum

```
sum=0;
for(int i=0; i<N; ++i)
  sum += A[i];
```

Example: Parallel Reduction



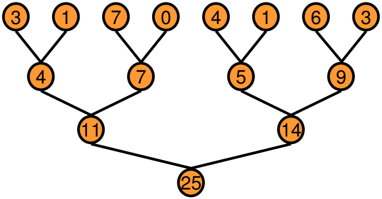
Vector sum

```
sum=0;
for(int i=0; i<N; ++i)
  sum += A[i];
```

Concurrent updates

- Need to protect accesses

Example: Parallel Reduction



Vector sum

```
sum=0;
for(int i=0; i<N; ++i)
  sum += A[i];
```

Concurrent updates

- Need to protect accesses

Idea:

- Synchronization within block is easy and fast
- Reduce in `__shared__` memory within block
- Serial reduction between blocks
- Most of the work parallelized

Example: Parallel Reduction

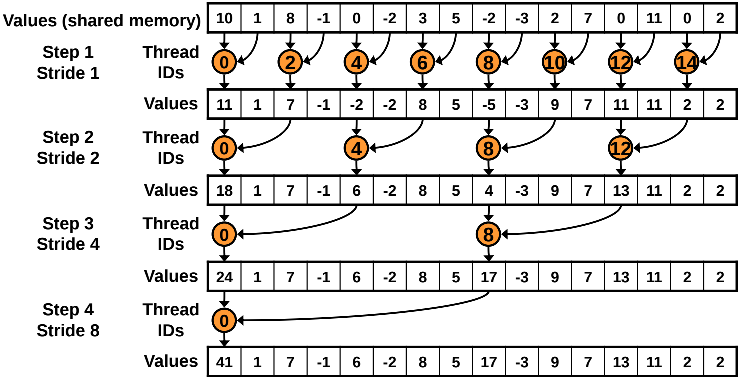
Kernel code:

```
__global__ void sum(float *res, float *v, int N)
{
  __shared__ float res_buf[BKSIZE];

  const int i_glob = threadIdx.x + blockIdx.x*blockDim.x;
  const int i_loc = threadIdx.x;

  res_buf[i_loc] = v[i_glob];
  __syncthreads();

  for(int s = 1; s<BKSIZE; s*=2) {
    if(i_loc % (2*s) == 0)
      res_buf[i_loc] += res_buf[i_loc+s];

    __syncthreads();
  }

  if(i_loc == 0) res[blockIdx.x] = res_buf[0];
}
```

Example: Parallel Reduction

What happens?

```
for(int s = 1; s<BKSIZE; s*=2) {
  if(i_loc % (2*s) == 0)
    res_buf[i_loc] += res_buf[i_loc+s];

  __syncthreads();
}
```



Example: Parallel Reduction

On the host:

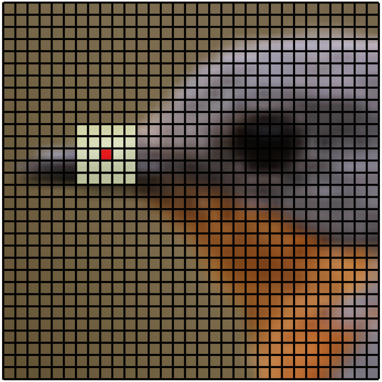
```
int nblocks = n/BKSIZE;
int nbytes = nblocks*sizeof(float);

// host and device arrays for partial results
number *res, *res_dev;

// allocation and initialization
res = malloc(nbytes);
cudaMalloc(&res_dev, nbytes);

// call kernel to compute partial sums
sum <<<nblocks,BKSIZE>>> (res_dev,v_dev,n);

// copy partial results to host
cudaMemcpy(res,res_dev,nbytes,
           cudaMemcpyDeviceToHost);

// serial reduction; sum up contributions from blocks
for(int i=1; i<nblocks; ++i)
  res[0] += res[i];
```

Question:
What ways do we have to synchronize threads in CUDA?

- Global barrier function
- Thread-block barrier function
- Kernel launch boundary
- Mutexes
- Atomic functions

Examples

Example: Gaussian blur



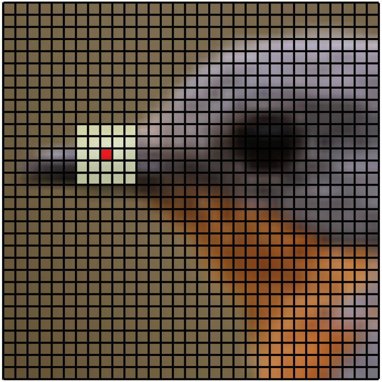
Blur an image:

- Average neighboring 5x5 pixel values

Algorithm:

```
for(int i=2; i<Height-2; i++)
  for(int j=2; j<Width-2; j++) {
    float tmp=0;
    for(int ii=-2; ii<=2; ii++)
      for(int jj=-2; jj<=2; jj++)
        tmp += Weight[(ii+2)*5+(jj+2)]
               *Ain[(i+ii)*Width + j+jj];
    Aout[i*Width + j] = tmp;
  }
```

Example: Gaussian blur



Blur an image:

- Average neighboring 5x5 pixel values

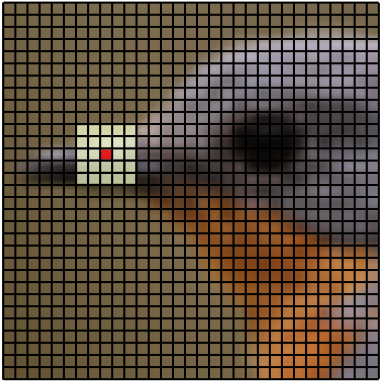
Algorithm:

```
for(int i=2; i<Height-2; i++)
  for(int j=2; j<Width-2; j++) {
    float tmp=0;
    for(int ii=-2; ii<=2; ii++)
      for(int jj=-2; jj<=2; jj++)
        tmp += Weight[(ii+2)*5+(jj+2)]
               *Ain[(i+ii)*Width + j+jj];
    Aout[i*Width + j] = tmp;
  }
```

Idea:

- All pixels are independent
- Both `i` and `j` loops are perfectly parallel

Example: Gaussian blur



Blur an image:

- Average neighboring 5x5 pixel values

Algorithm:

```
for(int i=2; i<Height-2; i++)
  for(int j=2; j<Width-2; j++) {
    float tmp=0;
    for(int ii=-2; ii<=2; ii++)
      for(int jj=-2; jj<=2; jj++)
        tmp += Weight[(ii+2)*5+(jj+2)]
               *Ain[(i+ii)*Width + j+jj];
    Aout[i*Width + j] = tmp;
  }
```

Idea:

- All pixels are independent
- Both `i` and `j` loops are perfectly parallel

Question: Which loop should we parallelize?

- The `i` loop
- The `j` loop
- Both loops

Example: Gaussian blur

Idea:

- On the GPU, want as many threads as possible
- One thread per pixel
- 2D blocks and grid

Example: Gaussian blur

Idea:

- On the GPU, want as many threads as possible
- One thread per pixel
- 2D blocks and grid

Kernel code:

```
__global__ void blur_kernel(float *Aout, float *Ain, float *W,
                            int width, int height)
{
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  int j = threadIdx.y + blockIdx.y*blockDim.y;

  if( (i>1 && i<height-2) && (j>1 && j<width-2) ) {
    float tmp = 0;

    for(int ii=-2; ii<=2; ii++)
      for(int jj=-2; jj<=2; jj++)
        tmp += W[(ii+2)*5+(jj+2)]*Ain[(i+ii)*width + j+jj];

    Aout[i*width + j] = tmp;
  }
}
```

Example: Gaussian blur

Host code:

```
// Aout, Ain, W are device arrays

// 2D grid and blocks
int nblocks_w = 1 + (width-1)/BKSIZE;  /* int division */
int nblocks_h = 1 + (height-1)/BKSIZE; /* rounding up */
dim3 gd_dim(nblocks_h,nblocks_w);
dim3 bk_dim(BKSIZE,BKSIZE);

blur_kernel<<<gd_dim,bk_dim>>>(Aout,Ain,W,width,height);
```

Example: Matrix-matrix product

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$$

Code:

```
for(int i=0; i<N; ++i)
  for(int j=0; j<N; ++j)
    for(int k=0; k<N; ++k)
      C[i*N+j] += A[i*N+k]*B[k*N+j]
```

Example: Matrix-matrix product

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$$

Code:

```
for(int i=0; i<N; ++i)
  for(int j=0; j<N; ++j)
    for(int k=0; k<N; ++k)
      C[i*N+j] += A[i*N+k]*B[k*N+j]
```

Idea:

- Computation of the elements in `C` are independent
- Both `i` and `j` loops are perfectly parallel
- Parallelize both loops!

Example: Matrix-matrix product

Parallelization:

- On GPU, want many threads
- Use one thread per element in C

Kernel code:

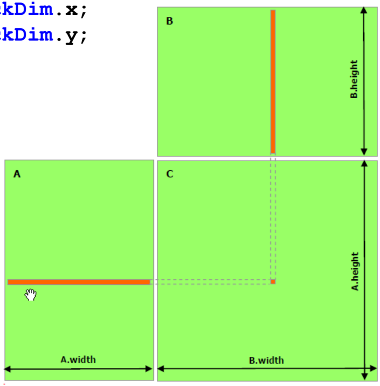
```
__global__ void matmul(float *C, float *A, float *B, int N) {
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  int j = threadIdx.y + blockIdx.y*blockDim.y;

  if(i < N && j < N)
    for(int k=0; k<N; k++)
      C[i*N+j] += A[i*N+k]*B[k*N+j];
}
```

Invocation:

```
const int n_blocks = 1+(N-1)/BKSIZE;
dim3 grid_dim(n_blocks,n_blocks); // grid and
dim3 block_dim(BKSIZE,BKSIZE);    // block dimensions

matmul<<<grid_dim,block_dim>>>(C_dev,A_dev,B_dev,N);
```

Example: Matrix-matrix product

```
__global__ void matmul(float *C, float *A, float *B, int N) {
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  int j = threadIdx.y + blockIdx.y*blockDim.y;

  if(i < N && j < N)
    for(int k=0; k<N; k++)
      C[i*N+j] += A[i*N+k]*B[k*N+j];
}
```

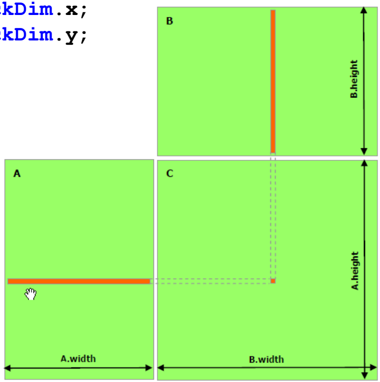


Problem:

- All threads with same `i` re-read whole `i`'th row of `A`
- Memory bandwidth wasted

Example: Matrix-matrix product

```
__global__ void matmul(float *C, float *A, float *B, int N) {
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  int j = threadIdx.y + blockIdx.y*blockDim.y;

  if(i < N && j < N)
    for(int k=0; k<N; k++)
      C[i*N+j] += A[i*N+k]*B[k*N+j];
}
```



Problem:

- All threads with same `i` re-read whole `i`'th row of `A`
- Memory bandwidth wasted

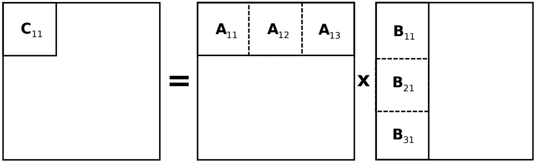
Q: How can we improve this?

- Cache data in `shared` memory
- Change data layout to improve coalescing

Example: Matrix-matrix product

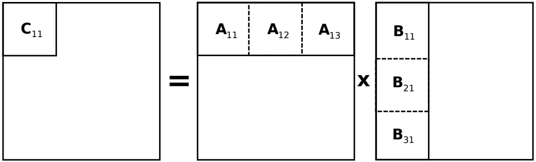
Tiled approach:

- E.g., 3x3 tiles
- Top left block:



$$C_{11} = A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}$$

Example: Matrix-matrix product



Tiled approach:

- E.g., 3x3 tiles
- Top left block:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}$$

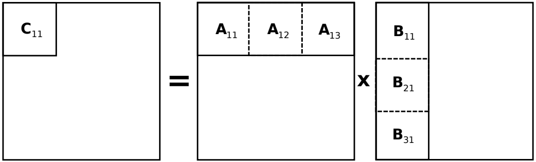
Algorithm:

- Load A_{11} and B_{11} into `shared` buffers
- Compute contribution from tiles
- Repeat with A_{12} and B_{21}
- Repeat with A_{13} and B_{31}

Example: Matrix-matrix product

Tiled approach:

- E.g., 3x3 tiles
- Top left block:



$$C_{11} = A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}$$

Algorithm:

- Load A_{11} and B_{11} into `shared` buffers
- Compute contribution from tiles
- Repeat with A_{12} and B_{21}
- Repeat with A_{13} and B_{31}

Motivation:

- Only have to read each tile from global memory *once per block*

Example: Matrix-matrix multiplication

```
__global__ void matmul(float *C, float *A, float *B, int N) {
  int i = threadIdx.x; int j = threadIdx.y;
  int iglob = threadIdx.x + blockIdx.x*blockDim.x;
  int jglob = threadIdx.y + blockIdx.y*blockDim.y;
  int ntiles = gridDim.x; // assume square grid
  // shared buffers
  __shared__ float atile[BKSIZE][BKSIZE];
  __shared__ float btile[BKSIZE][BKSIZE];

  for(int t=0; t<ntiles; ++t) {
    // load t'th tile of A and B into atile and btile
    atile[i][j] = /* element [i,j] of tile [blockIdx.x,t] of A */
    btile[i][j] = /* element [i,j] of tile [t,blockIdx.y] of B */
    // ensure data is ready
    __syncthreads();
    for(int k=0; k<BKSIZE; k++)
      C[iglob*N+jglob] += atile[i][k]*btile[k][j];
    // ensure all threads are done with data
    __syncthreads();
  }
}
```

For full solution, see GPU lab

Performance, tools and summary

Performance optimization

Parallelism:

- Use many small threads
- Avoid branch divergence

Performance optimization

Parallelism:

- Use many small threads
- Avoid branch divergence

Memory access:

- Coalesced memory access
- Use shared memory – avoid bank conflicts

Performance optimization

Parallelism:

- Use many small threads
- Avoid branch divergence

Memory access:

- Coalesced memory access
- Use shared memory – avoid bank conflicts

Data movement

- Avoid frequent host-device transfer
- Keep data on GPU

Using CUDA

Nvidia products with CUDA:

- Desktop / laptop graphics cards (GeForce)
- Dedicated compute cards – Tesla

Using CUDA

Nvidia products with CUDA:

- Desktop / laptop graphics cards (GeForce)
- Dedicated compute cards – Tesla

Compute Capability:

- GPUs support different features
- 1.3 – Double precision support
- 2.0 – L1 and L2 cache added
- 3.0 – Unified memory
- 3.5 – Dynamic parallelism (recursion)

Using CUDA

If you have an Nvidia GPU:

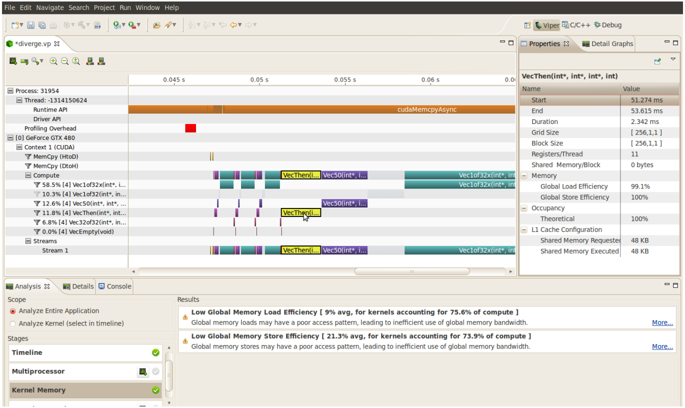
- Mac OS X / Linux:
  - Command line compilation / execution
  - `nvcc code.cu -o prog`
  - Specify architecture `-arch=sm_20` (For Compute Capability 2.0)
- Windows
  - Visual studio integration
- Installing:

  `docs.nvidia.com/cuda/cuda-quick-start-guide/index.html`

CUDA tools and libraries

Visual profiler

- Shows performance characteristics (cache usage, occupancy, compute/communication overlap, etc)
- Plugin for Eclipse / Visual studio



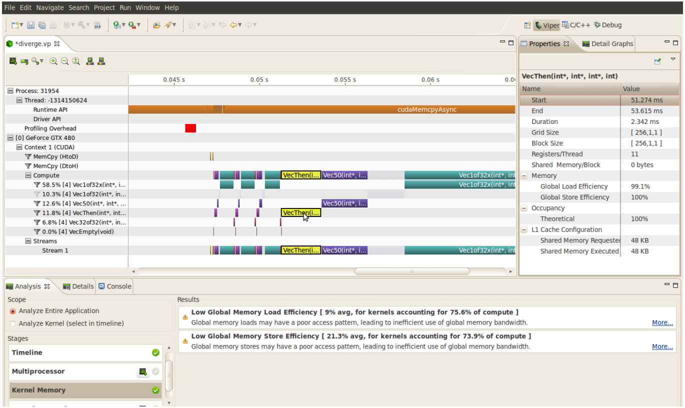
CUDA tools and libraries

Visual profiler

- Shows performance characteristics (cache usage, occupancy, compute/communication overlap, etc)
- Plugin for Eclipse / Visual studio

Also:

- Debugger `cuda-gdb`
- Memory checker `cuda-memcheck`



CUDA tools and libraries

Libraries:

- `cuBLAS` – Dense linear algebra
- `cuSPARSE` – Sparse linear algebra
- `cuFFT` – Fourier transforms etc
- `Thrust` – parallel data structures and algorithms – similar to C++ STL

CUDA tools and libraries

Libraries:

- `cuBLAS` – Dense linear algebra
- `cuSPARSE` – Sparse linear algebra
- `cuFFT` – Fourier transforms etc
- `Thrust` – parallel data structures and algorithms – similar to C++ STL

Avoid writing CUDA code:

```
// generate random data on the host
thrust::host_vector<int> h_vec(100);
// initialize h_vec with your own function init()
init(h_vec);
// transfer to device and compute sum
thrust::device_vector<int> d_vec = h_vec;

// binary operation used to reduce values
thrust::plus<int> binary_op;
// compute sum on the device
int sum = thrust::reduce(d_vec.begin(), d_vec.end(), 0, binary_op);
```

Hardware at UU

Computer Labs

- Rooms 2315, 1312, and 1313
- Windows machines with Nvidia GT630M (96 cores, 307 GFlop/s)

Hardware at UU

Computer Labs

- Rooms 2315, 1312, and 1313
- Windows machines with Nvidia GT630M (96 cores, 307 GFlop/s)

Uppmax

- Tintin cluster – 4 GPU nodes with Nvidia Tesla S2050 (1792 cores, 4.13 TFlop/s)
- Access in project