Exercise Session 10

Data Processing on Modern Hardware
263-3502-00L — Fall Semester 2012

Cagri Balkesen
cagri.balkesen@inf.ethz.ch

Department of Computer Science ETH Zurich, Switzerland

29 November 2012

Selective SQL Query on LINEITEM

Task: port column store implementation introduced in assignment 2 onto the GPU using CUDA or OpenCL. You will implement the following query:

```
SELECT SUM(quantity * extendedprice)
FROM   lineitem
WHERE  suppkey<Z
```

- Choose one of the following options for implementation
  - CUDA : if you have a CUDA-capable GPU (or on computers in student lab)
  - OpenCL : also can be executed on multi-core, cell etc.
  - Alternatively : device emulation mode (compiler option `-deviceemu`)

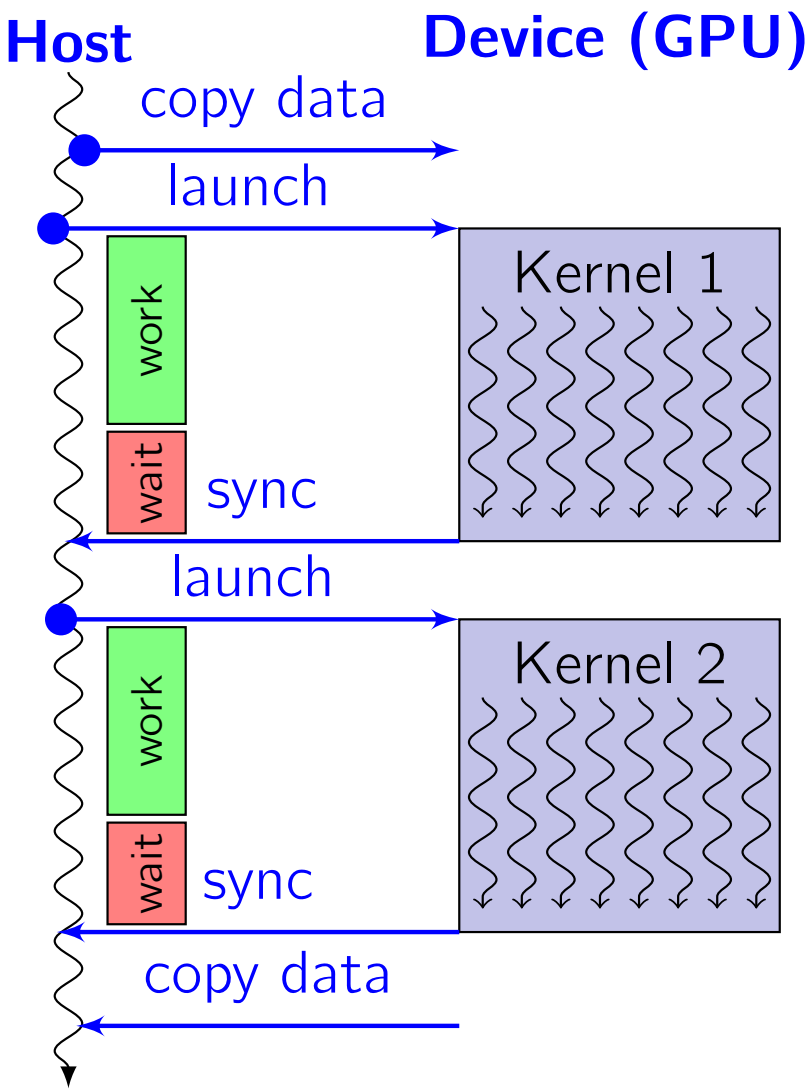
CUDA – Compute Unified Device Architecture

- Developed by NVIDIA
- High-level Programming Language for GPGPUs
  - C language with extensions
  - API and runtime environment
- PTX (Parallel Thread Execution) Virtual Machine
  - ISA for multiple GPU generation
  - JIT compilation to device code in driver

- For recent NVIDIA GPUs
- GeForce 8 Series onwards
- NVIDIA Tesla computing solution
  - GPUs without display output

source: NVIDIA

CUDA Computation Model

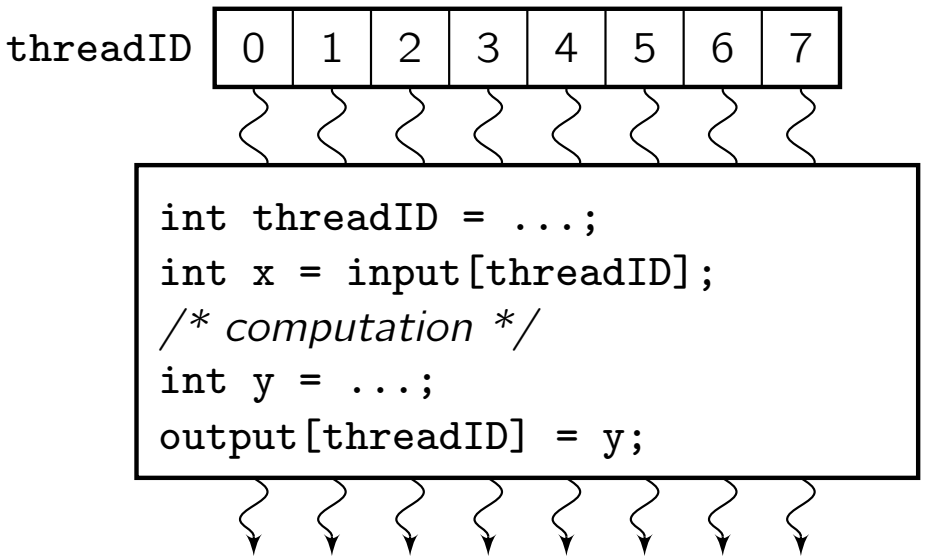


- Heterogeneous Processing on CPU and GPU
- CPU copies input data to GPU
- CPU launches compute kernels (each has many threads)
- CPU copies result data from GPU

- Kernel launches are asynchronous
- Memory copy is synchronous
- Explicit host synchronization operations, e.g., `cudaThreadSynchronize()`

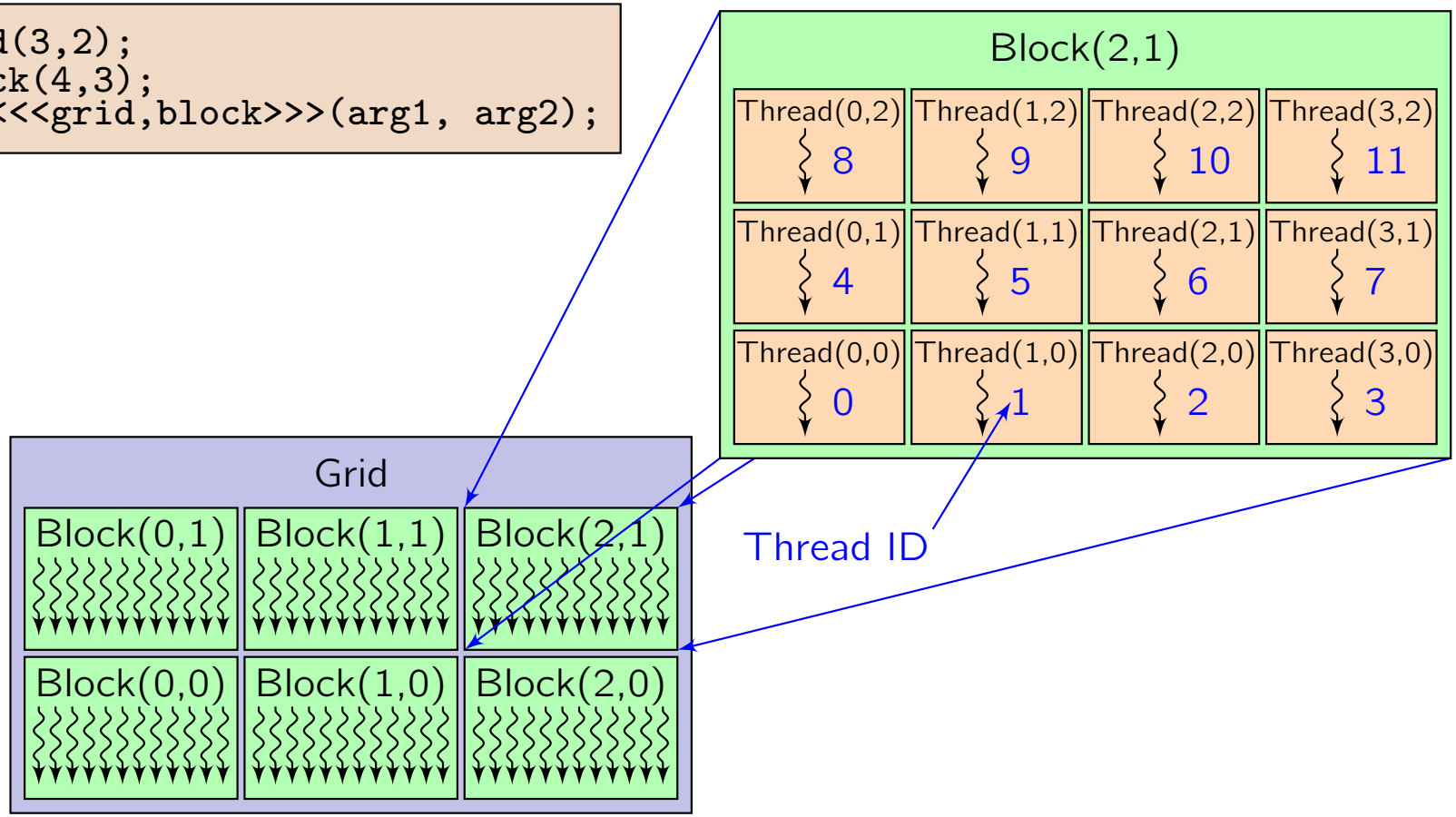
CUDA Parallel Threads

- Kernel code is executed by an **Array of Threads**
- All threads run the same code
- Threads have IDs
  - They use ID to figure out on which part of the data they have to work on.



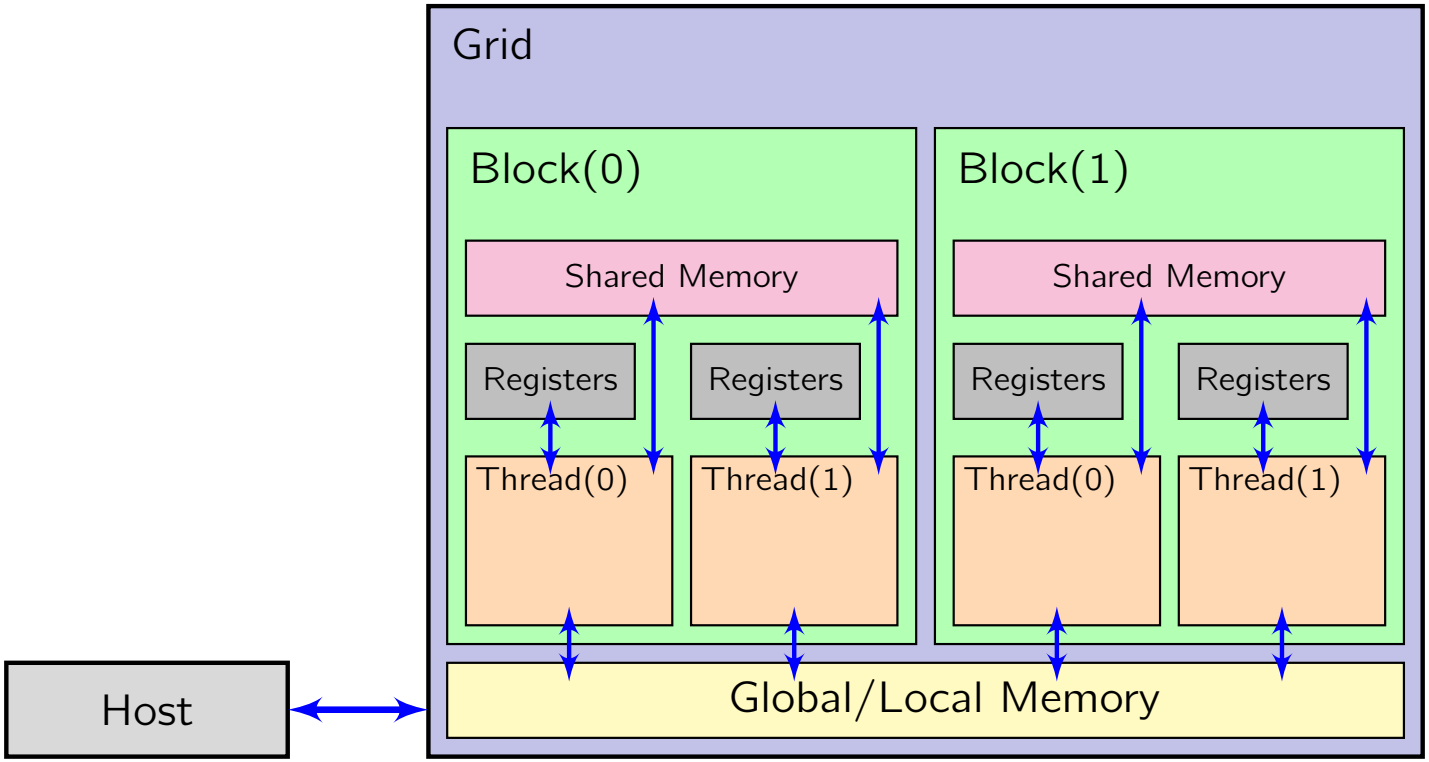
CUDA Thread Hierarchy

```
dim3 grid(3,2);
dim3 block(4,3);
myKernel<<<grid,block>>>(arg1, arg2);
```



- Block Index: 1- or 2-dimensional
- Thread Index: 1-, 2-, or 3-dimensional (x, y, z)
- Thread ID $= x + yD_x + zD_xD_y$ (block dimension: D_x, D_y, D_z)

Memory Access in Kernel



- **Shared Memory**
  - Shared among threads in a block
  - Small & fast on-chip memory (16 kB)
- **Global/Local Memory**
  - Large & uncached off-chip memory (up to 2 GB)
  - Host can only access global memory

CUDA Memories

Local Memory

- Thread-local
- Off-chip & uncached → expensive access

Shared Memory

- Shared by threads within block
- Fast on-chip, caching not necessary

Global Memory

- Accessible by host and all threads in all blocks
- Off-chip & uncached (no cache coherency required) → expensive access

Constant Memory

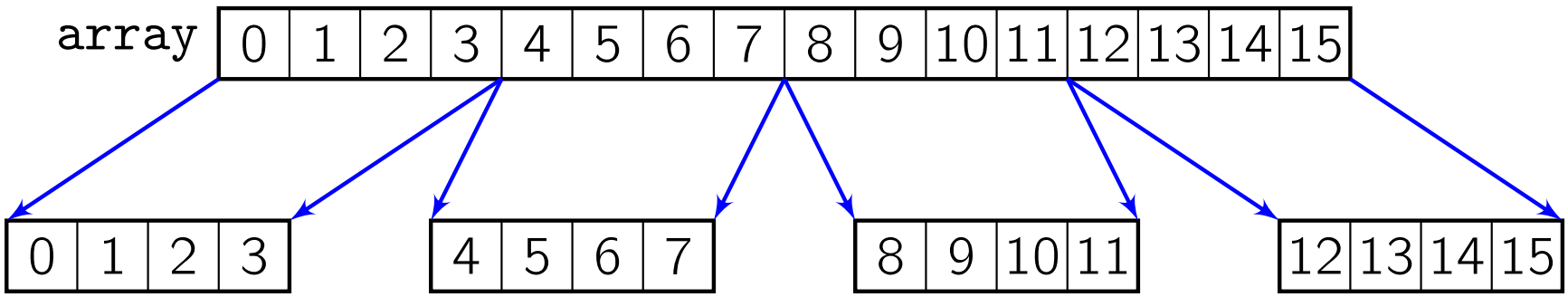
- Read-only from GPU
- Off-chip & cached (no cache coherency required, as read-only)

Texture Memory

- Read-only from GPU
- Off-chip & cached (no cache coherency required, as read-only)
- Optimized for 2D access (cache exploits 2D locality)
- Access through texture fetch

Increment Array Elements

Increment an N-element array `array` (embarrassingly parallel problem).
Split array in $\lceil N/B \rceil$ blocks of size B. For $N = 16$ and $B = 4$:




```
blockIdx.x=0          blockIdx.x=1          blockIdx.x=2          blockIdx.x=3
blockDim.x=4          blockDim.x=4          blockDim.x=4          blockDim.x=4
threadId.x=0,1,2,3    threadId.x=0,1,2,3    threadId.x=0,1,2,3    threadId.x=0,1,2,3
idx=0,1,2,3           idx=4,5,6,7           idx=8,9,10,11         idx=12,13,14,15
```

Use local **block** and **thread index** to determine global index in array:

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

CUDA : Increment Kernel

CUDA Kernel

```
__global__ void
incr(int *array_d, int N)
{
  int idx = blockIdx.x * blockDim.x
        + threadIdx.x;

  if (idx < N) {
      array_d[idx]++;
  }
}
```

- Directive `__global__` for symbols that are visible from both CPU and GPU code
- `blockIdx`, `blockDim`, and `threadIdx` are **built-in device variables** of type `dim3`
- The `if` is necessary if B does not divide N.
- Stored in CUDA `.cu` source file and using `nvcc`

CUDA: Launching the Kernel

increment.cu

```
__global__ void
incr(int *array_d, int N) {
  /* Kernel, see previous slide */
}

int main() {
  int N = 1000000;
  int blocksize = 256;
  int numblocks =
      (N+blocksize-1)/blocksize;

  /* allocate and init host memory */
  int *input_h =
      (int*)malloc(sizeof(int)*N);
  int *output_h =
      (int*)malloc(sizeof(int)*N);
  for (i=0;i<N;i++) input_h[i]=i;

  /* allocate device memory */
  int *array_d;
  cudaMalloc((void**)&array_d,
      sizeof(int)*N);

  /* copy from host to device */
  cudaMemcpy(array_d, input_h,
      N*sizeof(int),
      cudaMemcpyHostToDevice);

  /* launch kernel */
  dim3 dimBlock(blocksize);
  dim3 dimGrid(numblocks);
  incr<<<dimGrid,dimBlock>>>(array_d,N);

  /* copy from device back to host */
  cudaMemcpy(output_h, array_d,
      N*sizeof(int),
      cudaMemcpyDeviceToHost);
  ...
}
```

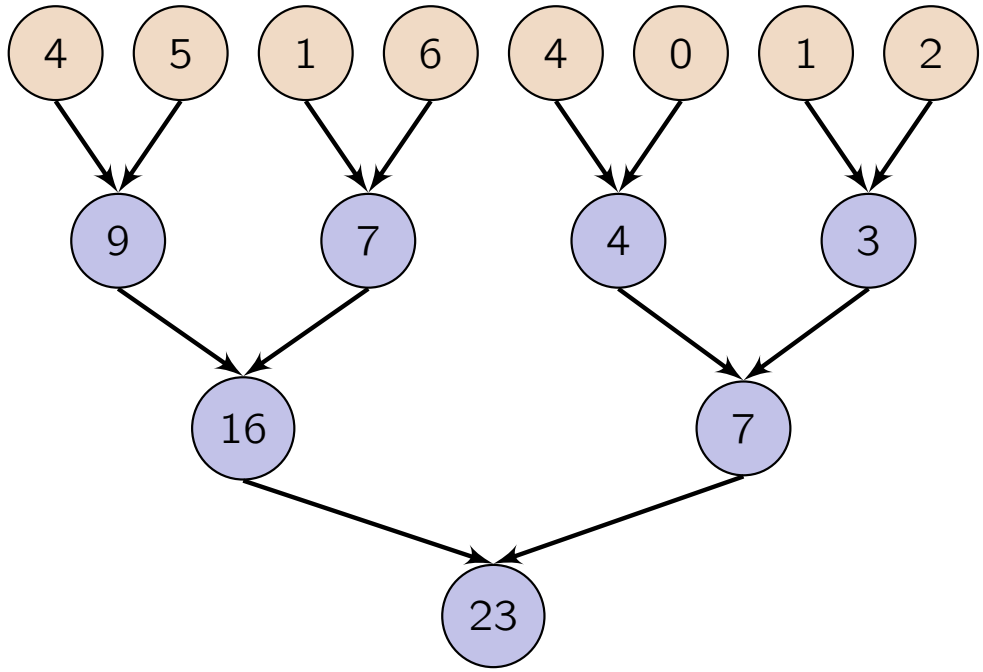
- 256 threads/block (for efficiency use ≥ 32)
- Suffix `_h` for host memory
- Suffix `_d` for device memory

Selective SQL Query on LINEITEM

```
SELECT SUM(quantity * extendedprice)
FROM   lineitem
WHERE  suppkey<Z
```

Parallel Data Reduction

- **Goal:** compute sum of array elements in parallel
- Use Case: "easy to implement — harder to get it right" (Mark Harris, NVIDIA)
- Example from NVIDIA CUDA SDK Sample Code
- Tree-based approach with concurrent operations
  - Level 1: 4 sums in parallel
  - Level 2: 2 sums in parallel
  - Level 3: 1 sum

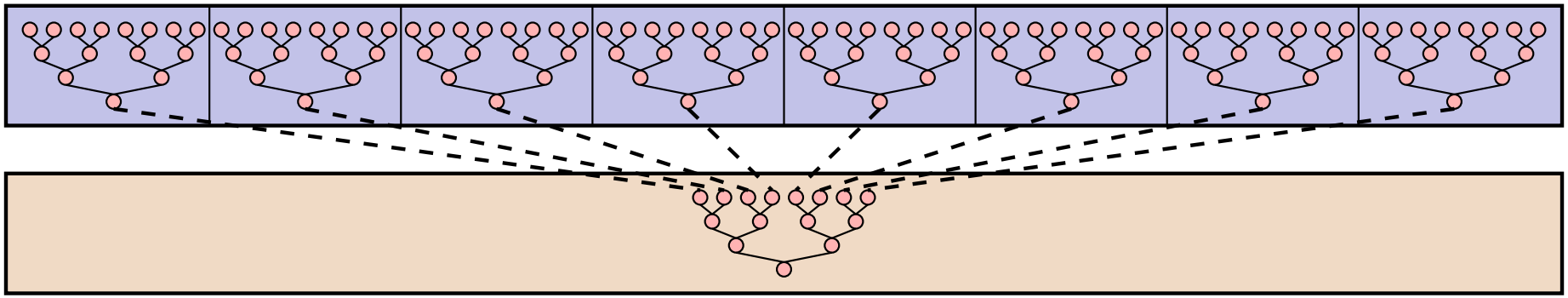


Synchronization

- Large array $\rightarrow$ many thread blocks to keep multiprocessors busy
- How to aggregate sum between threads?
  $\rightarrow$ **synchronization** required
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursive
- In CUDA, there is no **global** synchronization
- Do "global" synchronization implicitly by invoking multiple kernels, e.g., one per level.

Kernel Decomposition

level 1: 8 blocks



level 2: 1 block

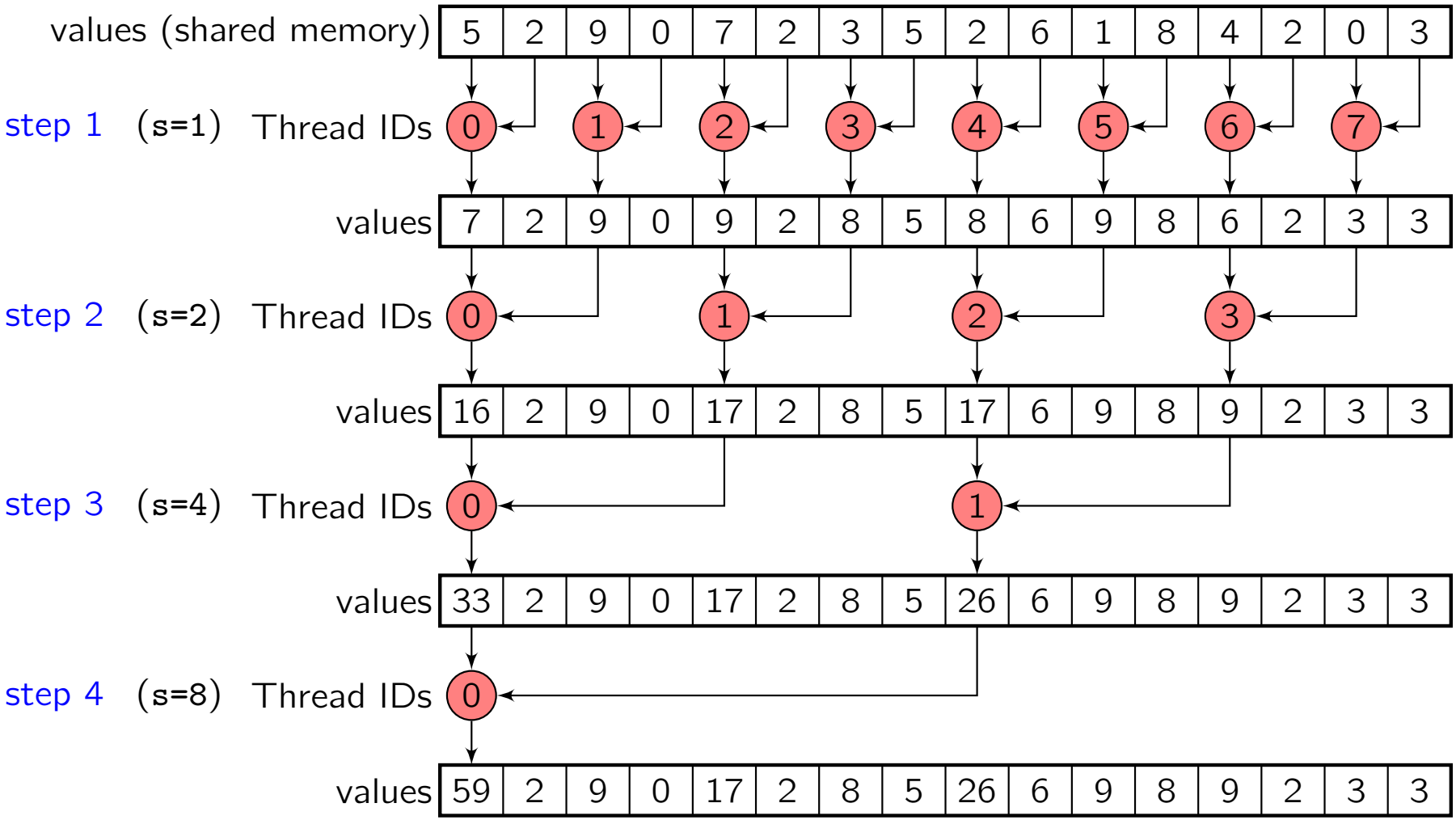
- Use multiple kernel invocations to synchronize computation
- Kernel code is the same for all invocations
- Simplified illustration: use at least 32 threads per block

Reduction Kernel — Implementation #1

```
__global__ void reduce(int *input, int *output) {
  extern __shared__ int sdata[];
  unsigned int tid = threadIdx.x; /* thread ID */

  /* index into data array for this thread */
  unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

  /* load data from global memory into shared memory */
  sdata[tid] = input[idx];
  __syncthreads(); /* wait for everybody else in the block */

  /* do the block-internal recursion */
  for (unsigned int s=1; s<blockDim.x; s *= 2) {
      int i = 2*s*tid;
      if (i < blockDim.x) sdata[i] += sdata[i+s];
      __syncthreads(); /* wait for everybody else in the block */
  }
  /* thread 0 writes result of sum into global memory */
  if (tid == 0) output[blockIdx.x] = sdata[0];
}
```

Implementation #1

For illustration `dimBlock.x=16`:



⇒ Strides 2, 4, and 8 result in 2-, 4- and 8-way bank conflicts.

Sequential Addressing — Implementation #2

Get rid of strided access

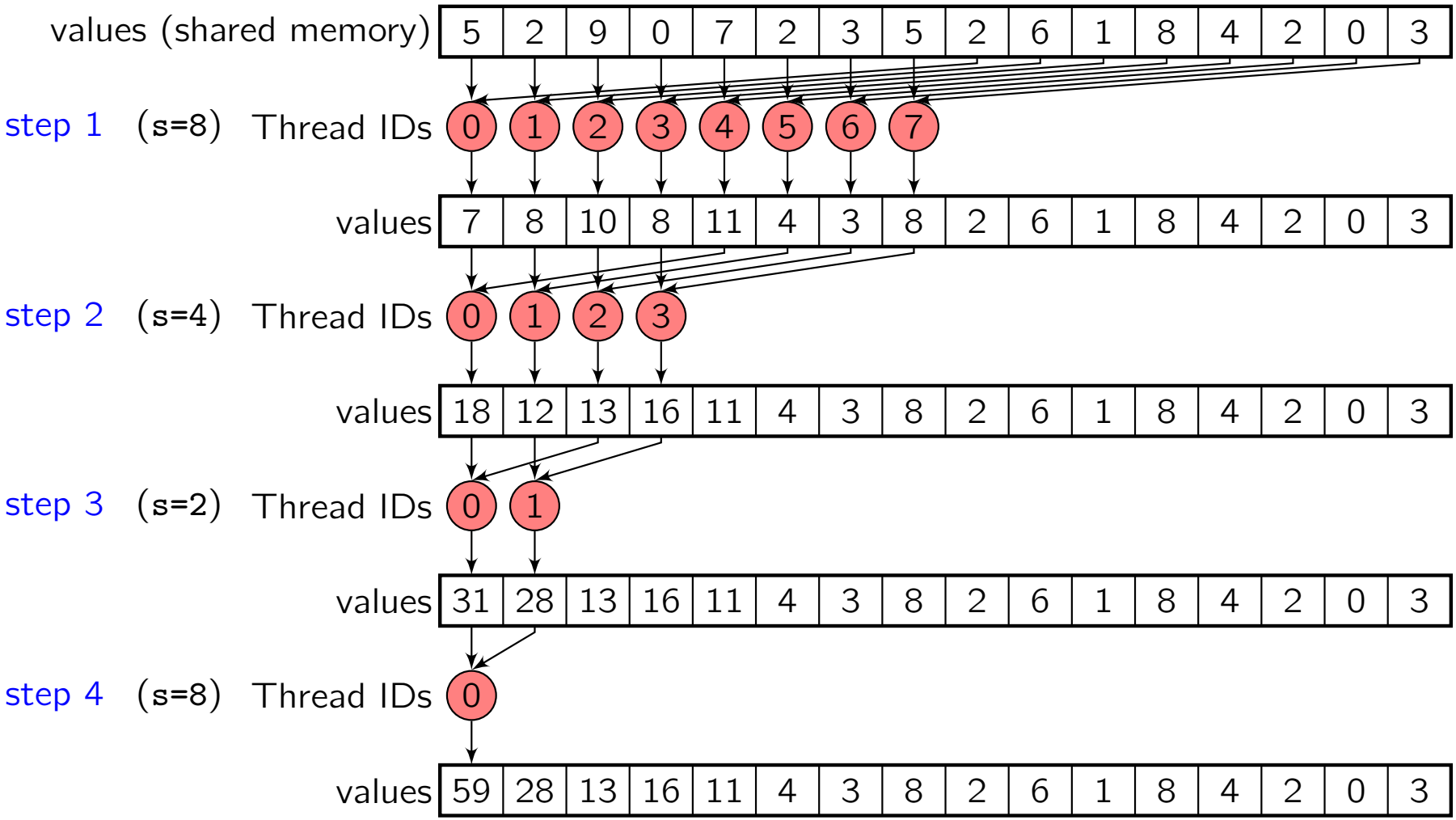
```
for (unsigned int s=1; s<blockDim.x; s *= 2) {
  int i = 2*s*tid;
  if (i < blockDim.x) {
      sdata[i] += sdata[i+s];
  }   __syncthreads(); /* wait for everybody else in the block */
}
```

by replacing it as:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
  if (tid < s) {
      sdata[tid] += sdata[tid+s];
  }
  __syncthreads(); /* wait for everybody else in the block */
}
```

Implementation #2

For illustration `dimBlock.x=16`:



⇒ Sequential addressing is conflict free.

Additional Optimizations

- *For details and more optimizations see documentation for NVIDIA CUDA SDK sample code,* `reduction` *example.*
- `http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf`

CUDA-GDB

- Documentation :
  `http://docs.nvidia.com/cuda/pdf/CUDA_GDB.pdf`
- CUDA-GDB is available with the CUDA Toolkit
- start debugger : `cuda-gdb cs_gpu`
- set breakpoint : `(cuda-gdb) break your-kernel`
- run program : `(cuda-gdb) run`
- peek into single thread:
  `(cuda-gdb) cuda block 4,0 thread 5,0,0`
- print variable x: `(cuda-gdb) print x`

Important:

- Before running CUDA-GDB compile your code with `-g -G` options and an optimization level of zero.
- Also shut down XServer (CTRL ALT F1) otherwise CUDA-GDB will complain.

lineitem.tbl too large?

- Last year some students experienced the following problem:

```
$ ./cs_gpu
CPU result : 209093448184600
test: FAIL
```

- What is the reason?
- How can we find the bug?

Safe Memory Allocation

```
void cudaSafe(cudaError_t error, const char* message) {
  if(error != cudaSuccess) {
    fprintf(stderr,
            "ERROR: %s :  %s \n",
            message,
            cudaGetErrorString(error));
    exit(EXIT_FAILURE);
  }
}

...

cudaSafe(cudaMalloc((void**)&input_d,
          numrows*sizeof(int32_t)),
          "cudaMalloc");
```

Safe Kernel Invocation

```
void cudaCheckError(const char *message) {
    cudaError_t error = cudaGetLastError();
    if (error != cudaSuccess) {
        fprintf(stderr,
            "ERROR: %s :  %s \n",
            message,
            cudaGetErrorString(error));
        exit(EXIT_FAILURE);
    }
}
...
yourkernel<<<dimGrid, dimBlock>>>(x,y);
cudaCheckError("Kernel Execution Failed!");
```

- Alternatively use `cutilSafeCall(...)` wrapper in `cutil_inline.h`

What was the problem?

- Not enough memory? → 6001152 rows.
  6001152×2×4 bytes ≈ 50MB
- Run
  ~/NVIDIA_GPU_Computing_SDK/C/bin/linux/release/deviceQuery

```
  Device 0:  "GeForce 9500 GT"
...
Total amount of global memory:  536150016 bytes
Maximum sizes of each dimension of a grid:  65535 x 65535 x 1
```

- blocksize = 64
- numblocks = (numrows+blocksize-1) ÷ blocksize
- numblocks = (6001152+63) ÷ 64 = 93768 ↯

Profiling CUDA Applications

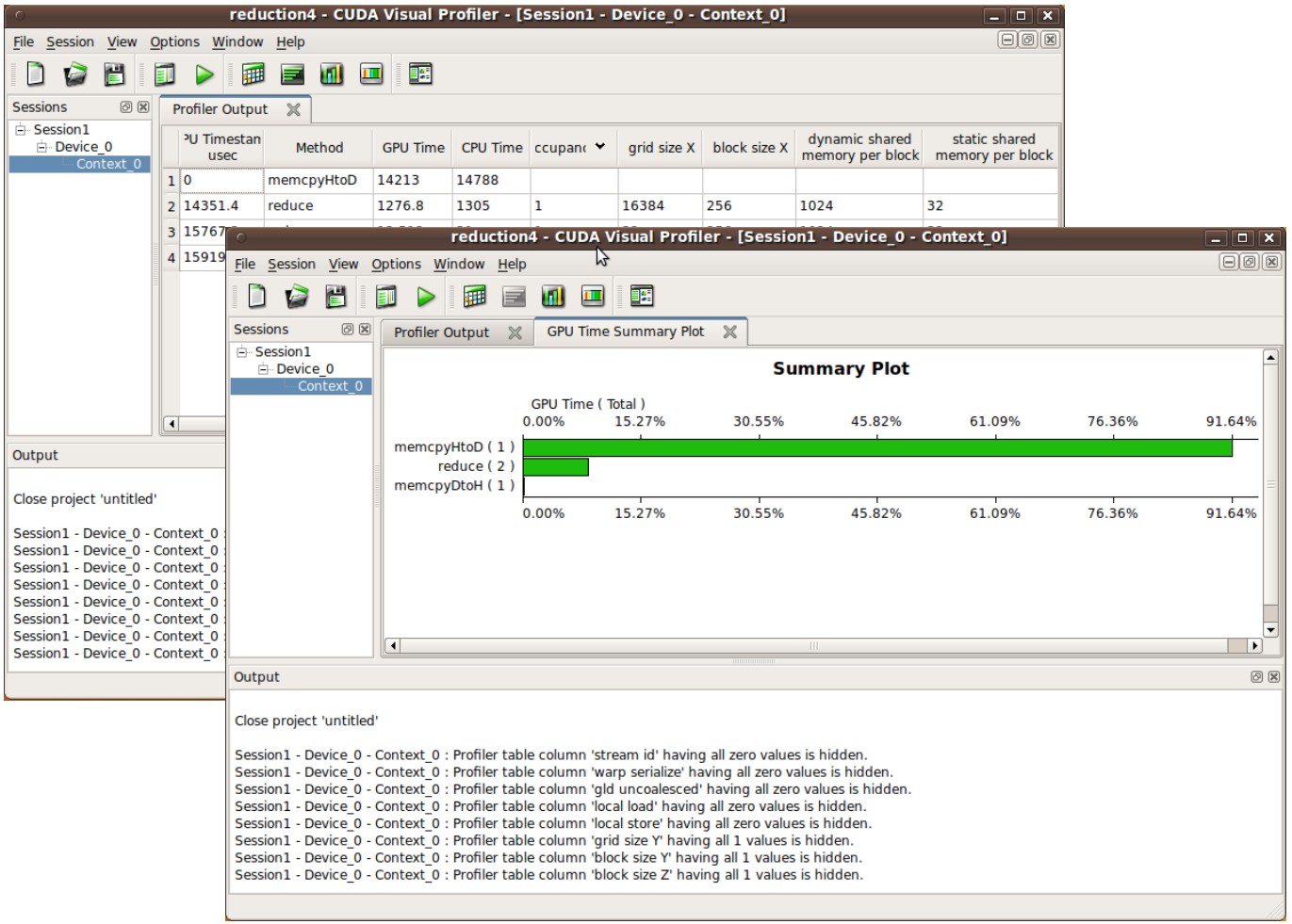
- Hardware Counters on-chip
  - Timestamps
  - Coalesced global memory load and stores
  - Local memory load and stores
  - Total branches and divergent (taken) branches
  - Instruction count
  - Number of serialized warps (memory conflicts)
  - Executed thread blocks

Control Profiler through environment variables

- `CUDA_PROFILE=0` or `1` to disable enable profiler → writes `cuda_profile.log`
- `CUDA_PROFILE_CONFIG` read profile configuration from file (up to four counters)
- For example, config.txt →
  `warp_serialize`
  `gridsize`
  ...

Visual Profiler

- Run `cudaprof`. QT Application



Resource Usage

- **Goal:** maximize occupancy of multiprocessors on GPU
- Occupancy depends on
  - Number of blocks and threads/block
  - Amount of local memory required
  - Register pressure of threads
  - Memory access pattern $\rightarrow$ use profiler
- Get more detailed from PTX assembler

```
$ nvcc --ptxas-options=-v increment.cu
ptxas info:  Compiling entry function '_Z7incrPii'
ptxas info:  Used 2 registers, 12+16 bytes smem
```

GPU Occupancy Calculator

- NVIDIA provides Excel spreadsheet to compute occupancy, opt. number of threads/block. Inputs:
  - Device, i.e., compute capability (1.0, 1.1, etc.)
  - Threads/block $\rightarrow$ from program
  - Registers/thread $\rightarrow$ from compiler
  - Shared Memory/block $\rightarrow$ from compiler

GPU Occupancy Calculator

CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	1.1	(Help)

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	2	
Shared Memory Per Block (bytes)	28	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	**768**	(Help)
Active Warps per Multiprocessor	**24**	
Active Thread Blocks per Multiprocessor	**3**	
Occupancy of each Multiprocessor	**100%**	

Physical Limits for GPU:	1.1
Threads / Warp	32
Warps / Multiprocessor	24
Threads / Multiprocessor	768
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	8192
Register allocation unit size	256
Shared Memory / Multiprocessor (bytes)	16384
Warp allocation granularity (for register allocation)	2

Allocation Per Thread Block	
Warps	8
Registers	512
Shared Memory	512

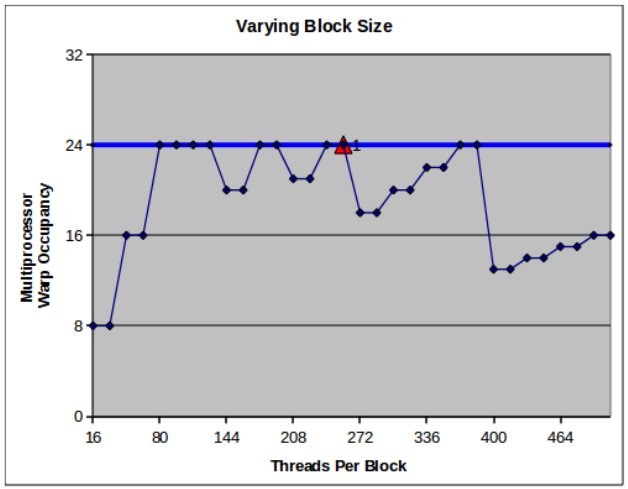
These data are used in computing the occupancy data in blue

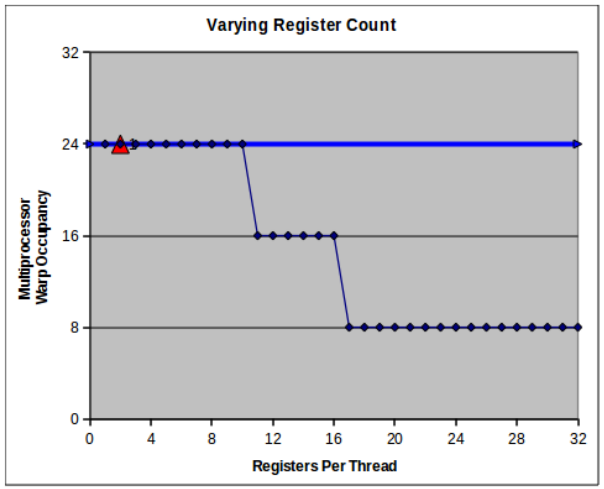
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	3
Limited by Registers / Multiprocessor	16
Limited by Shared Memory / Multiprocessor	32

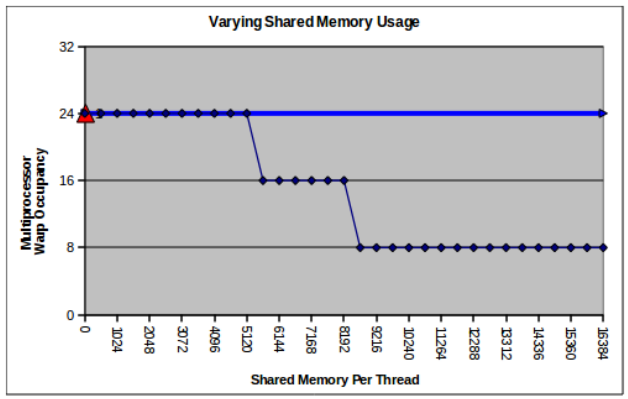
Thread Block Limit Per Multiprocessor highlighted RED

CUDA Occupancy Calculator	
Version:	1.5
Copyright and License	

Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.







Task

- Download code skeleton : `http://www.systems.ethz.ch/sites/default/files/file/dpmh_Fall2012/src-handout-06_tar.bz2`
- Study the *array increment* example (`example-cuda.cu` or `example-opencl.c`)
- Implement aforementioned query in `cs_gpu.cu`
- Until next week → try to get at least the *array increment* example running
- Note: performance results might not be great!

Hand in your Results

- E-Mail to cagri.balkesen@inf.ethz.ch
- Subject = DPMH:assignment6_{your netzname}
- Body = Description of Test System, e.g., CUDA & nVidia GeForce 9500 GT
- What speedup did you achieve?
- Attach source code (optional)