Gene Golub SIAM Summer School 2012 –
– Simulation and Supercomputing in the Geosciences

Parallel Computing on GPUs

Michael Bader, Alexander Breuer

Technische Universität München

References & Literature

- D. Kirk, W. Hwu:
  Programming Massively Parallel Processors, Morgan Kaufmann, 2010
- J. Sanders, E. Kandrot:
  CUDA by Example – An Introduction to General-Purpose GPU Programming, Addison Wesley, 2011
- NVIDIA CUDA Programming Guide

GPU Computing – Origins

Fixed-function graphics pipelines:

- 80ies/90ies: hardware configurable, but not programmable
- implementation of graphics APIs (OpenGL, DirectX, etc.)
- vertex shading/transform/lighting, raster operations, textures, etc.

Programmable Real-Time Graphics:

- shader programmability, floating-point pixel/shader/vertex processing
- resp. API extensions in DirectX, OpenGL
- programmable pipeline stages; hardware evolves towards massively parallel architectures

GPU Computing – Origins (2)

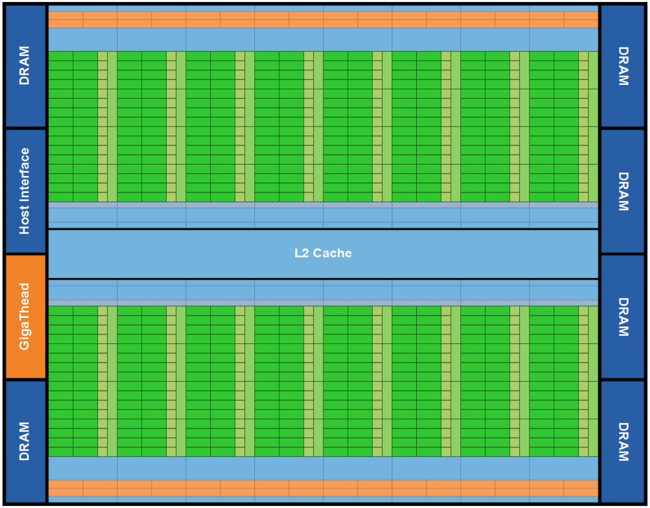
"GPGPU":

- general purpose computing on GPUs
- implement non-graphical algorithms/computations via shader functions
- driven by performance advantage of GPUs
  (for certain class of problems)

GPU Computing:

- hardware-side: general trend towards "many-core";
  GPUs evolve towards massively parallel, wider-purpose architectures
- software-side: programming models for GPU computing:
  CUDA, OpenCL, . . .

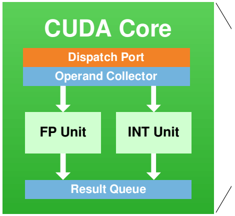
GPU Architectures – NVIDIA Fermi

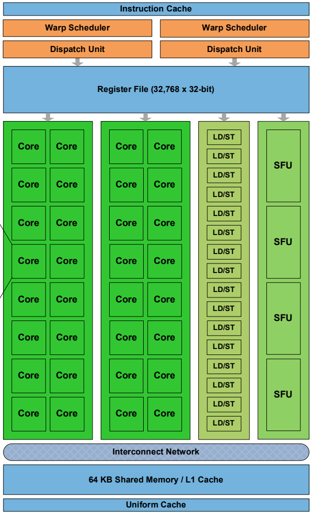


(source: NVIDIA – Fermi Whitepaper)

GPU Architectures – NVIDIA Fermi (2)

→ "Streaming Multiprocessor" (SM)





Fermi Streaming Multiprocessor (SM)

(source: NVIDIA – Fermi Whitepaper)

GPU Architectures – Some First Observations

Processing Units

- massive parallelism: lots (hundreds) of cores
- Instruction Cache, Warp Scheduler, Dispatch Unit:
  $\rightarrow$ shared for multiple CUDA cores
  $\rightarrow$ "single instruction multiple thread" concept

Memory:

- on-chip DRAM (GPU main memory)
  $\rightarrow$ accounts for memory bandwidth values
- L2 cache $\rightarrow$ fairly recent development for GPUs
- L1 cache/shared memory and registers:
  $\rightarrow$ again shared between many cores

“Moving/Accessing Memory is Evil”

Development of Memory Performance

- CPU performance increases by 59 % each year
- memory bandwidth increases by 23 % each year
- memory latency improves by 5 % each year

⇒ memory is a bottleneck!

Additional Issues:

- access of individual elements vs. “blocks” of data
- stream-like access to memory

⇒ same problems for CPUs and GPUs
⇒ certain differences in approaching these problems
(GPU approach: latency hiding by fast task switching)

Part I

CUDA Basics

CUDA – Architecture Model

Host & Device:

- host = regular CPU, main memory
- device(s) = GPU/coprocessor(s) with separate memory

Parallel Computing Concept:

- massively parallel architecture (hundreds of cores)
- lightweight threads, hardware-supported;
  typically multiple threads assigned to every core
- massive parallelism hides memory latency:
  → hardware-supported (fast) task-switching
  → focus on data parallelism & vectorization

CUDA – Programming Model

CUDA as extension of C/C++:

- host code (program control) and device code (GPU) combined in a single C program
- device code consists of massively parallel **kernels** that are off-loaded to the GPU
- language extension for defining and calling kernels
- API functions to allocate device/host memory, synchronize threads, etc.
- SIMD/SPMD (single instruction/program, multiple data)??
  $\rightarrow$ SIMT (single instruction/multiple thread)

Let’s Dive Into an Example: Matrix Multiplication

A parallel algorithm for matrix multiplication:

```
for i from 1 to n do in parallel
    for k from 1 to n do in parallel
        for j from 1 to n do
            C[i,k] += A[i,j]*B[j,k]
```

- algorithmic model (“PRAM”): execute on n^2 processors $\rightarrow$ compute the elements C[i,k] entirely in parallel
- CUDA: n^2 CUDA threads; each thread executes one j-loop (i.e., computes one element C[i,k])
- part 1: memory transfer (host$\rightarrow$device and device$\rightarrow$host)
- part 2: launch/execution of kernel code for each j-loop

Matrix Multiplication – Memory Allocation

```
__host__
void matrixMult(float *A, float *B, float *C, int n)
{
   int size = n*n*sizeof(float);
   float* Ad; float* Bd; float* Cd;

   cudaMalloc((void**)&Ad, size);
   cudaMalloc((void**)&Bd, size);
   cudaMalloc((void**)&Cd, size);

   /*-- missing here:                               --*/
   /*-- transfer of input/output to/from GPU --*/
   /*-- matrix multiplication on GPU          --*/

   cudaFree(Ad); cudaFree(Bd); cudaFree(Cd);
}
```

Matrix Multiplication – Memory Transfer

```
__host__
void matrixMult(float *A, float *B, float *C, int n)
{
   int size = n*n*sizeof(float);
   float* Ad; float* Bd; float* Cd;
   /*-- cudaMalloc for Ad, Bd, Cd skipped --*/

   cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
   cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
   cudaMemcpy(Cd, C, size, cudaMemcpyHostToDevice);

   /*-- perform multiplication on device --*/

   cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
   /*-- cudaFree for Ad, Bd, Cd skipped --*/
}
```

Matrix Multiplication – CUDA Kernel

```
__global__
void matrixMultKernel(float* Ad, float* Bd, float* Cd, int n)
{
   /*-- determine i and k for the current thread: --*/
   int i = threadIdx.x;
   int k = threadIdx.y;

   /*-- compute a single element of C[i,k]: --*/
   float Celem = 0;
   for(int j=0; j<n; j++) {
      float Aelem = Ad[i*n+j];
      float Belem = Bd[j*n+k];
      Celem += Aelem*Belem;
   };
   Cd[i*n+k] += Celem;
}
```

Kernel Invocation: Grids and Blocks

```
__host__
void matrixMult(float *A, float *B, float *C, int n)
{
   /* ... */
   dim3 dimBlock(n,n);
   dim3 dimGrid(1,1);
   matrixMultKernel<<<dimGrid,dimBlock>>>(Ad,Bd,Cd,n);
   /* ... */
}
```

- threads (max. 512) are combined to 3D **blocks**:
  $\rightarrow$ threadIdx.x, threadIdx.y, threadIdx.z
  (example above: $n \times n \times 1$ block)
- blocks are combined to 1D or 2D **grids**:
  $\rightarrow$ blockIdx.x, blockIdx.y

Grids and Blocks in CUDA

Blocks:

- threads can be organised as 1D, e.g. (128,1,1), 2D, e.g. (16,16,1), or 3D, e.g. (4,8,16) blocks
- limited to 512 threads per block
- threads in one block are always executed in parallel
- and can use separate, shared memory

Grids:

- dim3, but 2D layout (3rd component ignored)
- up to 65536 $\times$ 65536 blocks per grid
- blocks in a grid may be executed in parallel (but, in practice, will be scheduled to available cores)

Matrix Multiplication – with Grid

```
__global__
void matrixMultKernel(float* Ad, float* Bd, float* Cd, int n)
{
   /*-- determine i and k for the current thread: --*/
   int i = blockIdx.x * TILE_SIZE + threadIdx.x;
   int k = blockIdx.y * TILE_SIZE + threadIdx.y;

   /*-- compute a single element of C[i,k]: --*/
   float Celem = 0;
   for(int j=0; j<n; j++) {
      float Aelem = Ad[i*n+j];
      float Belem = Bd[j*n+k];
      Celem += Aelem*Belem;
   };
   Cd[i*n+k] += Celem;
}
```

Matrix Multiplication – with Grid (2)

```
__host__
void matrixMult(float *A, float *B, float *C, int n)
{
    /* ... */
    dim3 dimBlock(TILE_SIZE,TILE_SIZE);
    dim3 dimGrid(n/TILE_SIZE,n/TILE_SIZE);
    matrixMultKernel<<<dimGrid,dimBlock>>>(Ad,Bd,Cd,n);
    /* ... */
}
```

- choose TILE_SIZE = 16
  (“square” blocks and number of threads $<$ 512)
- in practice: requires padding of matrix (i.e., fill with zeros) to match size (multiple of 16)
- works for large matrices – how about performance?

Part II

Hardware-Aware Programming with CUDA: GPU Memory

CUDA Memory

Types of **device** memory in CUDA:

- per thread: **registers** and **local memory**
  → locally declared variables and arrays (local memory),
  → lifetime: kernel execution
- per block: **shared memory**
  → keyword __shared__,
  → lifetime: kernel execution
- per grid: **global memory** and **constant memory**
  → keywords __device__, __constant__;
  → lifetime: entire application
- vs.: CPU main memory (host memory)

Matrix Multiplication – Performance Estimate

Multiplication kernel:

```
for(int j=0; j<n; j++) {
   float Aelem = Ad[i*n+j];
   float Belem = Bd[j*n+k];
   Celem += Aelem*Belem;
};
```

- memory bandwidth: 11.2 GB/s on my laptop, NVIDIA NVS4200M)
  vs. 97.6 GB/s (on mathgpu, Tesla C2070)
- two floating-point operations (multiply and add) per two floating-point variables (each 4 byte)
- thus: max. of 3 giga float variable can be transferred from global memory per second
- limits performance to $< 3 GFlop/s$
- **Experiment:** profiling of matrix multiplication kernel ($\rightarrow$ NVIDIA Visual Profiler)

Matrix Multiplication and Memory Usage

- observation: simple matrix multiplication kernel is slow (far below peak performance)
- anticipated reason: only access to slow global memory; performance limited by memory bandwidth between global memory and CUDA cores

Matrix Multiplication with Tiling

Remedy: **Tiling**

- switch to tile-oriented implementation:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{pmatrix}$$

- copy matrix tiles A_{11}, B_{11}, etc., into shared memory
- let all threads of a block work together on shared tile
  $\rightarrow$ multipliy matrix tiles: $A_{11}B_{11}$, $A_{12}B_{21}$, $\ldots$
- accumulate result tile back on matrix in global memory:
  $C_{11} = A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}$
  $C_{12} = A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32}$

Matrix Multiplication – with Tiles

```
__global__
void matrixMultKernel(float* Ad, float* Bd, float* Cd, int n)
{
   __shared__ float Ads[TILE_SIZE][TILE_SIZE];
   __shared__ float Bds[TILE_SIZE][TILE_SIZE];
   int tx = threadIdx.x;
   int ty = threadIdx.y;
   int i = blockIdx.x * TILE_SIZE + tx;
   int k = blockIdx.y * TILE_SIZE + ty;
   for(int m=0; m < n/TILE_SIZE; m++) {
      Ads[tx][ty] = Ad[ i*n + m*TILE_SIZE+ty];
      Bds[tx][ty] = Bd[ (m*TILE_SIZE+tx)*n + k];

      /* perform matrix multiplication on shared tiles */
```

Matrix Multiplication – with Tiles

```
    /* (cont.) */
    for(int m=0; m < n/TILE_SIZE; m++) {
        Ads[tx][ty] = Ad[ i*n + m*TILE_SIZE+ty];
        Bds[tx][ty] = Bd[ (m*TILE_SIZE+tx)*n + k];
        __syncthreads();
        /* perform matrix multiplication on shared tiles */
        for(int j=0; j<TILE_SIZE; j++)
            Celem += Ads[tx][j]*Bds[j][ty];

        __syncthreads();
    };
    Cd[i*n+k] += Celem;
}
```

A Note on Synchronisation

Barrier synchronisation in CUDA:

- not all threads of a thread block necessarily executed in parallel (remember, e.g., fast switching between different task groups)
- between parallel/collective reads and subsequent writes $\Rightarrow$ **barrier required** for all threads within a thread block:

  `__syncthreads();`

- all threads need to execute (or not) the same(!) call to `__syncthreads()`
- threads of the same block scheduled to the same hardware unit
- in contrast: no synchronisation features for threads in a grid $\rightarrow$ reason: *transparent scheduling* of entire blocks

Updated Performance Estimate

- at start, each thread loads one matrix element from global memory
- shared memory $\rightarrow$ no further loads in TILE_SIZE m-iterations
- we reduce the memory transfer from global memory to $1/\text{TILE\_SIZE}$
- for TILE_SIZE = 16: new performance limit at $640 GFlop/s$

$\rightarrow$ we've eliminated a major bottleneck, but apparently hit another …

Experiment: profiling of tiled matrix multiplication kernel

Part III

Hardware-Aware Programming with CUDA: Threads, Warps, and Coalesced Access

Thread Assignment and Thread Scheduling

CUDA programming model:

- 2D grid of 3D blocks
- blocks are scheduled to execution resources (block-by-block; parallel, if possible)

CUDA hardware:

- cores (streaming processors, SP) organised into *streaming multiprocessors* (SM)
- multiple blocks can be simultaneously assigned to an SM

Thread Scheduling:

- threads of a block are sub-divided into **warps** (32 threads)
- warps are scheduled to SMs; threads in a warp are executed in SIMT (single instruction, multiple thread) fashion

Coalesced Memory Access

Hardware issues in memory access:

- DRAM (i.e., global memory) built, such that multiple *contiguous* memory slots are read together (compare: cache lines)
- in CUDA: memory access of threads will be simultaneous by all threads in a warp
- if accesses are to contiguous memory locations (in the order given by the thread ID) and 16-word-aligned
  $\Rightarrow$ **coalesced** memory access
- coalesced memory access required to achieve full memory bandwidth

Coalesced Access in Matrix Multiplication

Copy matrix tile *m* from global into shared memory:

```
Ads[tx][ty] = Ad[ i*n + m*TILE_SIZE+ty];
Bds[tx][ty] = Bd[ (m*TILE_SIZE+tx)*n + k];
__syncthreads();
```

Do we have coalesced memory access?

- row computation: i = blockIdx.x * TILE_SIZE + tx
- column computation: k = blockIdx.y * TILE_SIZE + ty

$\Rightarrow$ stride-1 access w.r.t. ty, stride-n access w.r.t. tx

Question: which threads are combined in a warp?

Warps and Coalesced Access

Combination of threads into warps:

- 1D thread block: thread 0, …31 into warp 0; thread 32, …63 into warp 1; etc.
- 2D thread block: x-dimension is “faster-running”; e.g.:
  - dimBlock(8,8,1), i.e., 64 threads (2 warps)
  - then threads (0,0), …, (7,3) are in warp 0 and threads (0,4), …, (7,7) are in warp 1
- 3D thread block: x, then y, then z

Tile-Copying in Matrix Multiplication:

- threads with consecutive tx value in one warp
- leads to stride-n access $\Rightarrow$ **not coalesced**

Matrix Multiplication with Coalesced Access

Switch tx and ty ⇒ stride-1 (coalesced) access to Ad, Bd:

```
int i = blockIdx.y * TILE_SIZE + ty;
int k = blockIdx.x * TILE_SIZE + tx;
float Celem = 0;
for(int m=0; m < n/TILE_SIZE; m++) {
   Ads[ty][tx] = Ad[ i*n + m*TILE_SIZE+tx];
   Bds[ty][tx] = Bd[ (m*TILE_SIZE+ty)*n + k];
   __syncthreads();
   for(int j=0; j<TILE_SIZE; j++)
      Celem += Ads[ty][j]*Bds[j][tx];
   __syncthreads();
};
Cd[i*n+k] += Celem;
```

Experiment: profiling of “coalesced” matrix multiplication kernel

Memory Latency for Tile Transfers

Recapitulate tiled matrix multiplication:

- tiles of 16×16 matrix elements
  $\rightarrow 16^2 = 256$ threads per tile (also per thread block)
- thus: 8 warps (32 threads each)
- examine load operation for matrix tiles

```
Ads[ty][tx] = Ad[ i*n + m*TILE_SIZE+tx];
Bds[ty][tx] = Bd[ (m*TILE_SIZE+ty)*n + k];
__syncthreads();
```

  $\rightarrow$ delay due to memory latency
- all threads in a warp wait for data to arrive
- but another warp can be scheduled to work

Question: are there enough warps to hide latency?

Tiled Matrix Multiplication with Prefetching

Include prefetching of blocks to reduce “idle” time for memory transfer:

1. load first tile into register(s)
2. copy register(s) to shared memory
3. load next tile into register(s)
4. compute current tile
5. proceed with 2 (if there are more tiles)
6. compute last tile

(see code example)

Experiment: profiling of “prefetching” matrix multiplication kernel

Use of Registers

How many registers do we actually need?

- two registers per thread $\rightarrow$ 512 registers per block
- GTX 285: each SM has 1024 thread slots
  $\Rightarrow$ 4 blocks can be run in parallel
- hence, we require 2024 additional registers

How many registers are there?

- GTX 285: 8192 registers for each SM
- dynamical partitioning of registers to blocks/threads
- assume that a multiplication kernel requires 10 registers:
  $10 \cdot 16^2 = 2560$ registers per block
  $\Rightarrow$ only 3 blocks can run

Dynamic Partitioning of Resources

Limits for execution of threads (for NVS 4200M)

- threads per block (1024)
- blocks per SM (8)
- thread slots per SM (1024)
- registers per SM (32768)
- padding of warps (blocks per warp not a multiple of 32)

Leads to trade-offs for performance:

- limited number of threads reduces parallelism (and, thus, achievable performance)
- “performance cliffs”: slight change in set-up might lead to jumps in available blocks
- requires detailed estimates (or lots of testing?) to determine best option

Towards High-Performance Matrix Multiplication

More Options for Optimisation:

- loop unrolling (save loop instructions and address arithmetics)
- thread granularity: compute 1×2 or 1×4 blocks per thread (requires to load Ads or Bds only once)
- how do different optimisations interact with resource limitations (available registers, etc.)

Thread Execution – SIMT

“Single Instruction, Multiple Thread”:

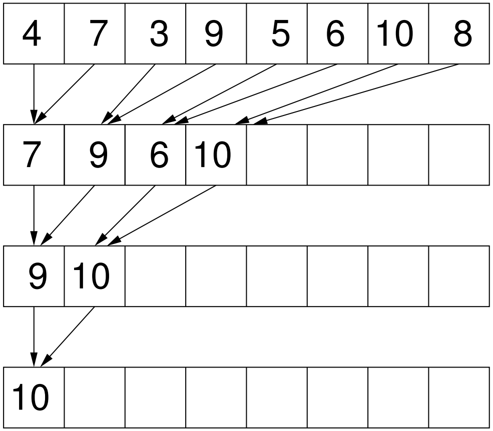
- hardware executes same instruction for all threads in a warp
- execution of if-else-statements:
  - first, execute if-branch (for part of the threads)
  - then, execute else-branch (for all other threads)
- “diverging” thread execution; similar: for- and while-loops

Exercise: Maximum Reduction

- task: parallel computation of the maximum
- reduction operation via “binary fan-in”
- how can diverging threads be avoided?

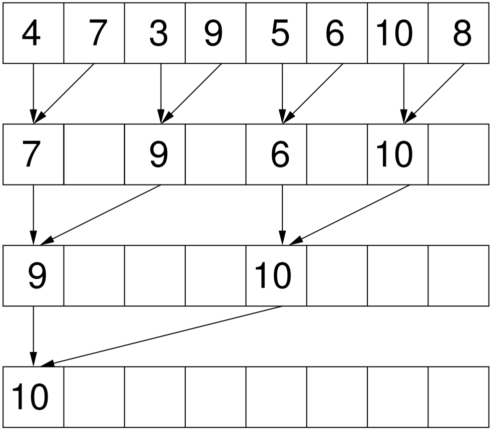
Maximum Search – Parallel Reduction

Classical approach: “Binary Fan-In”

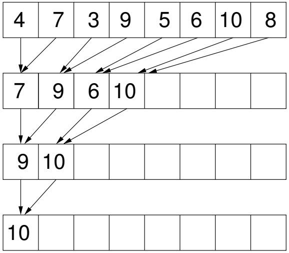


Maximum Search – Parallel Reduction (2)

Alternative approach for “Binary Fan-In”

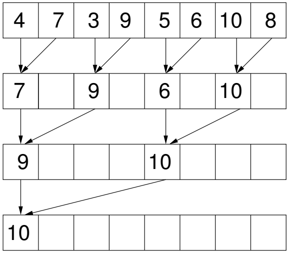


"Binary Fan-In" and Warps



Alternative 1 on GPU:

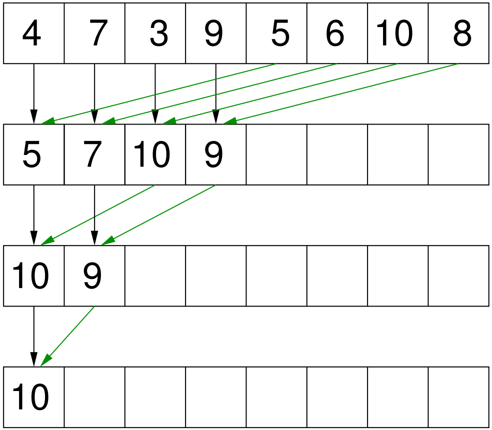
- in-place computation:
  "local" maxima overwrite data
- with warps:
  no control on sequence of updates



Alternative 2 on GPU:

- in-place computation and
  synchronisatzion of warps
  no longer critical, but:
- strided access within warps:
  $\rightarrow$ idle threads/"thread divergence"

Maximum Search – “Binary Fan-In” for GPUs



“Contiguous” threads (of a warp) access contiguous memory cells!

Maximum Computation on the GPU

Basic Implementation:

```
__global__
void kernelMaximum(float* h, int size) {
  /*-- size has to be a power of 2 and smaller than 512/1024 --*/
   int tx = threadIdx.x;
   for (int i=size/2; i>0; i=i/2) {
       __syncthreads();
       if (tx < i) {
           if( h[tx] < h[tx+i] ) h[tx] = h[tx+i];
        };
   };
}
/*-- kernel call: --*/
dim3 dimBlock(threads); // 1D thread block
kernelMaximum<<<dimGrid,dimBlock>>>(h,threads);
```

Maximum on the GPU

Towards Optimized Implementations

Further steps to consider:

- extend for larger thread blocks (multiple kernel calls, e.g.)
- perform reduction in shared memory
  (copy/reduce to shared memory in first step)
  $\rightsquigarrow$ perhaps a version that does not overwrite the original array . . .
- unrolling of loops, optimize for specific thread block sizes

Reference and optimized kernel:

M. Harris: Optimizing Parallel Reduction in CUDA (NVIDIA tutorial)

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

Part IV

GPU Computing: A Call to Arms?

What We Did Not Cover . . .

Other programming models:

- **OpenCL**: programming model/language for hybrid computing; backends for GPUs and CPUs
  → short code example (Ch. Dittrich):

```
queue.enqueueNDRangeKernel(shallowWatersEulerTimeStepKernel,
    cl::NullRange, cl::NDRange(MATRIX_SIZE, MATRIX_SIZE),
    cl::NDRange(TILE_SIZE, TILE_SIZE), 0, &e );
```

- **OpenACC**: compiler directives to specify loops/parallel code that can be offloaded to accelerator hardware → short code example (source: NVIDIA):

```
!$acc parallel loop
    do i=0, n-1
        !--> loop body skipped
    end do
!$acc end parallel loop
```

What We Did Not Cover . . .

Other programming models (cont.):

- **ArBB**: (Intel Array Building Blocks, unfortunately deprecated . . .)
  → short code example (D. Gudu):

```
void eulerTimestep_map(
    arbb::f32& h, arbb::f32& hu, arbb::f32& hv,
    /*-- further parameters --*/ ) {
    hu = arbb::select(h<1e-05f, 0.0f,
              hu - dt *( (Fhu-neighbor(Fhu,0,-1))/dx
                       + (Ghu-neighbor(Ghu,-1,0))/dy + Bx/dx ));
}
/*-- corresp. kernel call: --*/
arbb::map(eulerTimestep_map)(arbb_h, arbb_hu, arbb_hv, /*-- --*/);
```

  ⇝ better vectorization of code

A List of Further “Usual Suspects”

Further parallel programming models/languages in HPC:

- MPI, OpenMP are still around . . .
- PGAS languages (“partitioned global address space”): Unified parallel C, Coarray Fortran, Chapel, Titanium, X10, . . .
- Intel Parallel Building Blocks (TBB, Cilk, Array Notations, . . .) and **#pragma** simd constructs

The Quest for the Future Parallel Programming Model

- MPI will probably stay for quite some time . . .
- MPI+X to replace MPI+OpenMP for hybrid parallelization?
- extensions of C/C++/Fortran: not clear which exactly, but there will certainly be some . . .
- **guaranteed:** you need to program/think in parallel

Twelve Ways to Fool the Masses

(selection)

- Quote only 32-bit performance results, not 64-bit results.

  $\rightarrow$ single precision on the GPU, double precision on the CPU??

- Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application.

  $\rightarrow$ did you include the transfer time from/to GPU?

- Compare your results against scalar, unoptimized code on Crays

  $\rightarrow$ compare optimized GPU code with non-optimized CPU code

Source: D. H. Baily: *Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers*, Supercomputing Review, 1991

Twelve Ways to Fool the Masses (2)

(continued)

- When direct run time comparisons are required, compare with an old code on an obsolete system

  → **compare performance on the latest GPU with a single-core CPU??**

- Quote performance in terms of processor utilization, parallel speedups or MFLOPS per dollar

  → **MFLOPS per Watt!!**

- Mutilate the algorithm used in the parallel implementation to match the architecture

  → **Gauß-Seidel on GPU vs. Multigrid on CPU??**

- If all else fails, show pretty pictures and animated videos, and don't talk about performance

  → **Well, that's more or less what we did :-)**