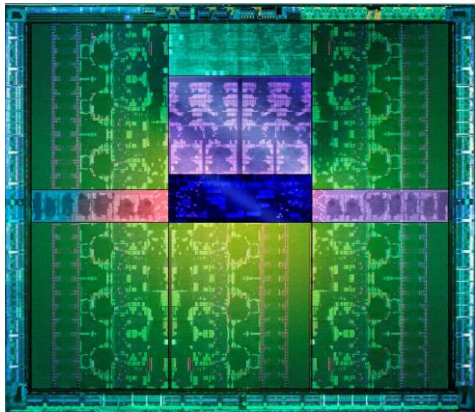
GPU programming

Dr. Bernhard Kainz

BioMedIA
Imperial College London



Overview



- About myself
- Motivation
- GPU hardware and system architecture
- GPU programming languages
- GPU programming paradigms
- Example program
- Memory model
- Tiling
- Reduction
- State-of-the-art applications

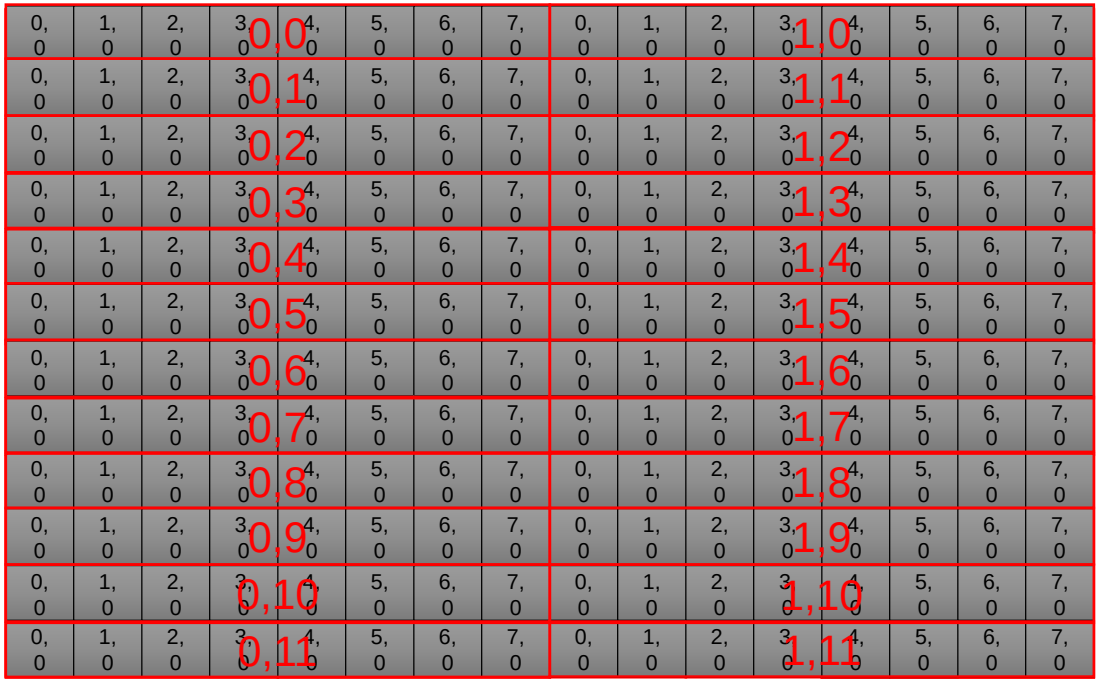
Last week

This week

Distinguishing between threads



- blockId and threadId



2D Kernel example



- using **threadIdx** and **blockIdx** execution paths are chosen
- with **blockDim** and **gridDim** number of threads can be determined

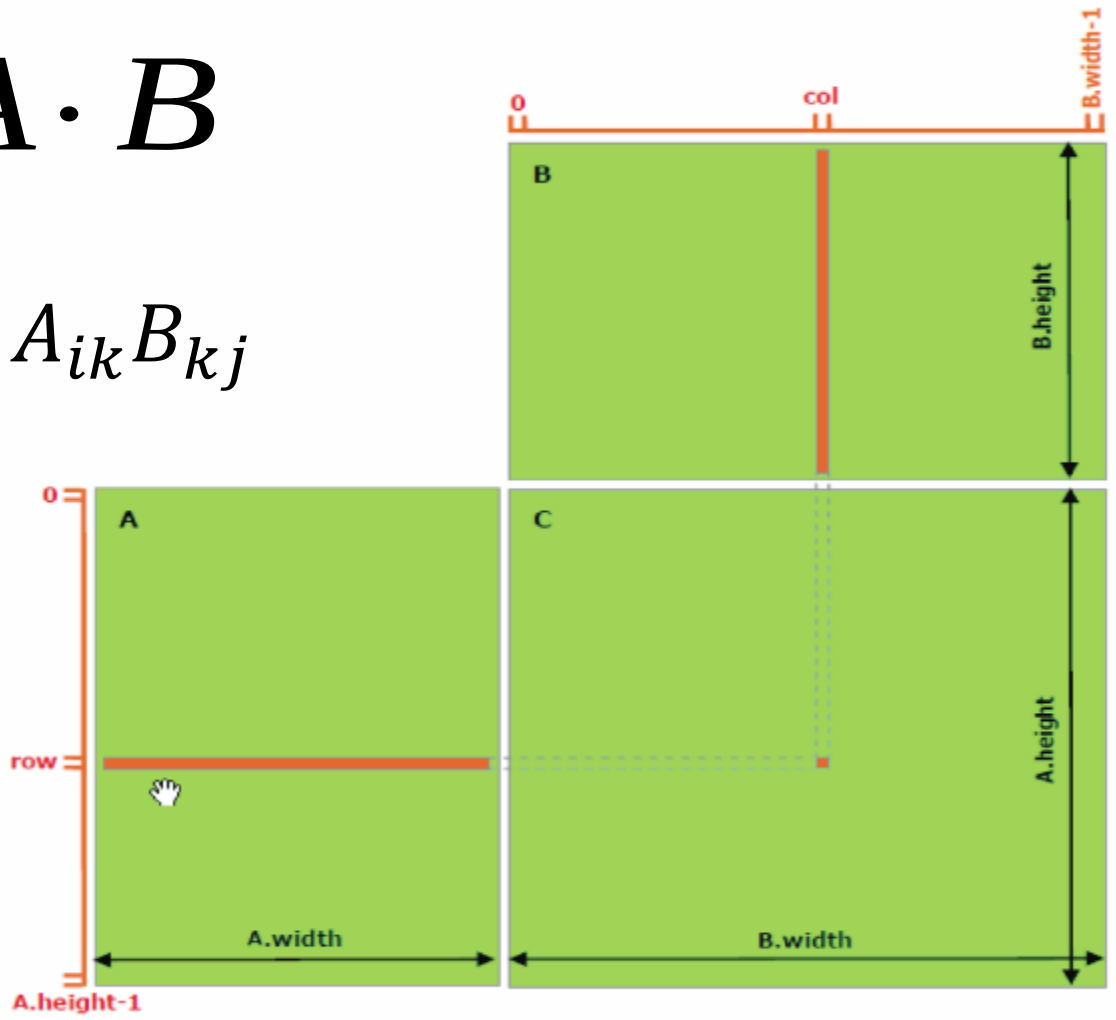
```
__global__ void myfunction(float *input, float* output)
{
  uint bid = blockIdx.x + blockIdx.y * gridDim.x;
  uint tid = bId * (blockDim.x * blockDim.y) +
             (threadIdx.y * blockDim.x) + threadIdx.x;
  output[tid] = input[tid];
}

dim3 blockSize(32,32,1);
dim3 gridSize((iSpaceX + blockSize.x - 1)/blockSize.x,
              (iSpaceY + blockSize.y - 1)/blockSize.y), 1)
myfunction<<<gridSize, blockSize>>>(input, output);
```

Matrix Multiplication Example

$$C = A \cdot B$$

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



Matrix Multiplication Example



```
7  std::vector<float> cpu_multiply(const std::vector<float>& A, size_t A_cols, size_t A_rows,
8    const std::vector<float>& B, size_t B_cols, size_t B_rows)
9  {
10   std::cout << "computing C on the CPU";
11   size_t C_rows = A_rows,
12          C_cols = B_cols;
13   std::vector<float> C(C_rows*C_cols);
14
15   PointInTime t0;
16   size_t elements = A_cols;
17   auto cIt = C.begin();
18   for(size_t y = 0; y < C_rows; ++y)
19     for(size_t x = 0; x < C_cols; ++x)
20     {
21       float c_xy = 0;
22       for(size_t i = 0; i < elements; ++i)
23         c_xy += A[y*A_cols + i]*B[i*B_cols + x];
24       *cIt++ = c_xy;
25     }
26   PointInTime t1;
27   std::cout << " done in " << (t1-t0) << " seconds\n";
28   return C;
29 }
```

Loop-based parallelism

Matrix Multiplication Example



```
82  __global__ void d_multiply(const float* A, int A_cols, int A_rows,
83                             const float* B, int B_cols, int B_rows,
84                             float* C)
85  {
86      int C_cols = B_cols,
87          C_rows = A_rows;
88
89      int x = blockIdx.x*blockDim.x + threadIdx.x;
90      int y = blockIdx.y*blockDim.y + threadIdx.y;
91
92
93      int elements = A_cols;
94      float c_xy = 0;
95      for(int i = 0; i < elements; ++i)
96          c_xy += A[y*A_cols + i]*B[i*B_cols + x];
97      C[y*C_cols + x] = c_xy;
98
99  }
```

```
for(size_t y = 0; y < C_rows; ++y)
  for(size_t x = 0; x < C_cols; ++x)
  {
```

Matrix Multiplication Example



```
float* A = new float[A_rows*A_cols];
float* B = new float[B_rows*B_cols];
float* C = new float[B_cols*A_rows];
//some matrix initialization
float* d_A, d_B, d_C;

cudaMalloc((void**)&d_A, A_rows*A_cols*sizeof(float));
cudaMalloc((void**)&d_B, B_rows*B_cols*sizeof(float));
cudaMalloc((void**)&d_C, B_cols*A_rows*sizeof(float));
cudaMemcpy(d_A, A, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, cudaMemcpyHostToDevice);

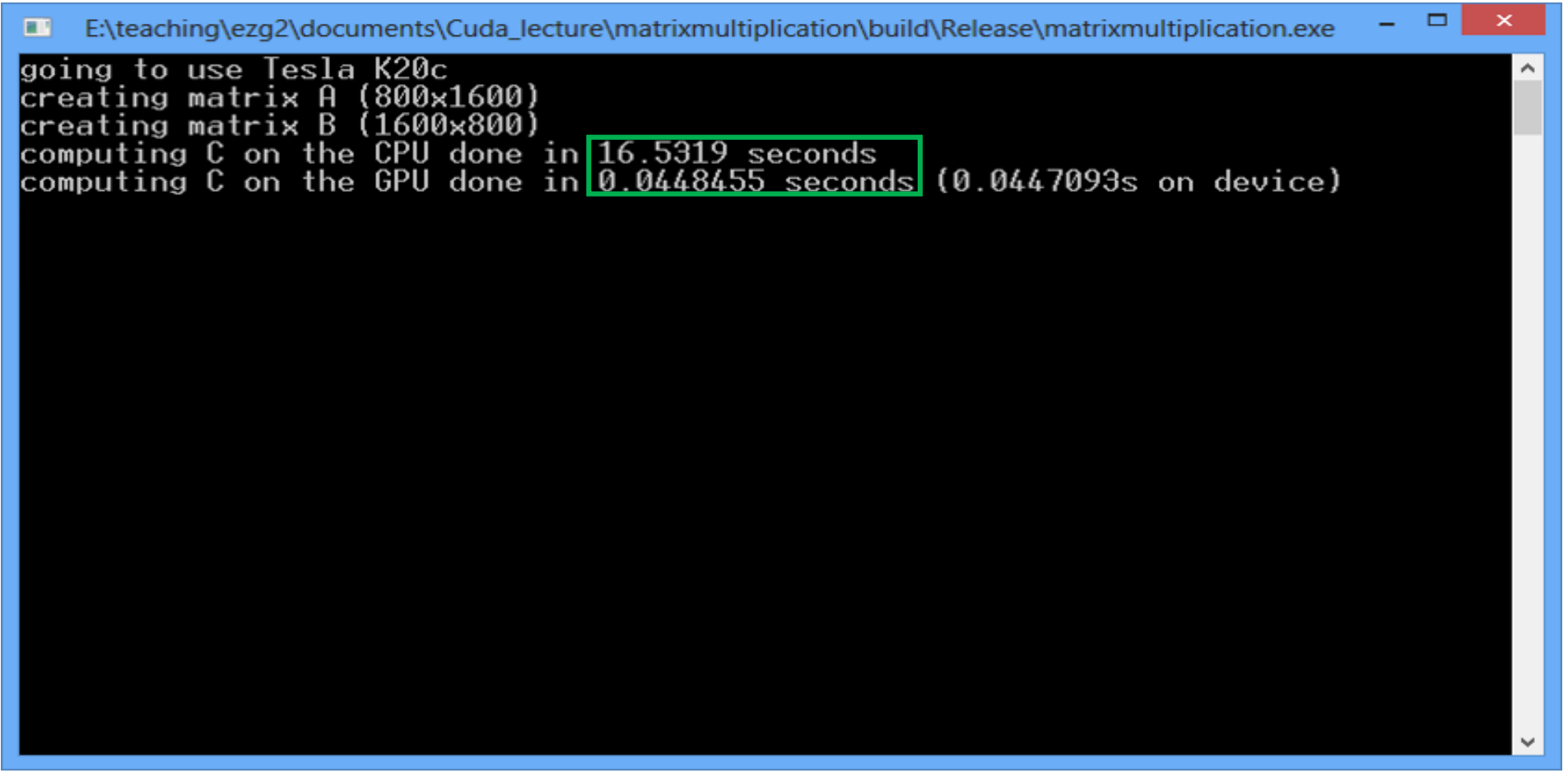
//THE KERNEL CALL
dim3 blockSize(16,16,1);
dim3 gridSize( C_cols/blockSize.x, C_rows/blockSize.y, 1);
d_multiply<<<gridSize, blockSize>>>(d_A, A_cols, A_rows, d_B, B_cols, B_rows, d_C);

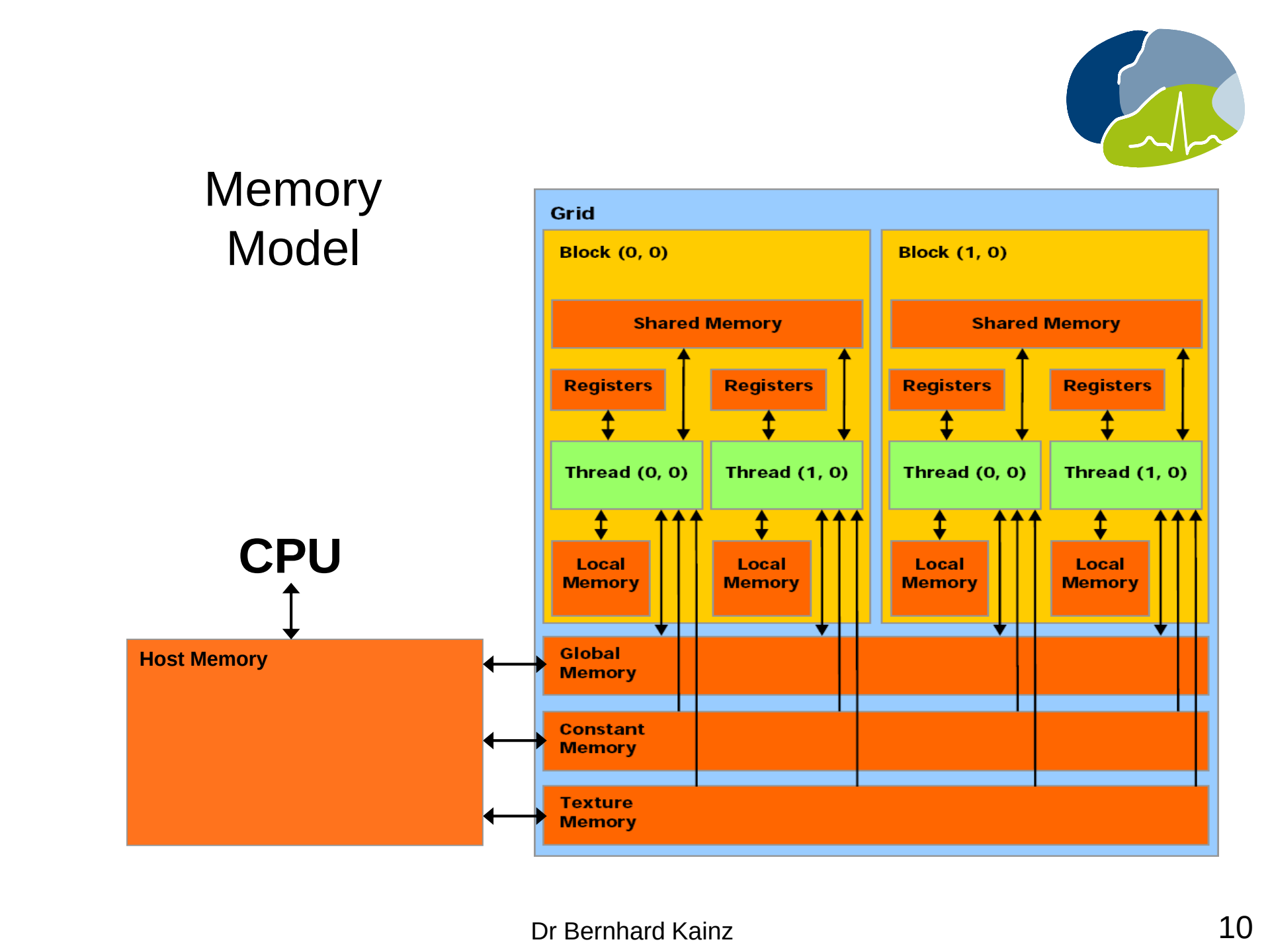
//SYNCHRONIZE
cudaError err = cudaDeviceSynchronize();

cudaMemcpy(C, d_C, cudaMemcpyDeviceToHost);
//free stuff
```

Matrix Multiplication Example







Memory Model

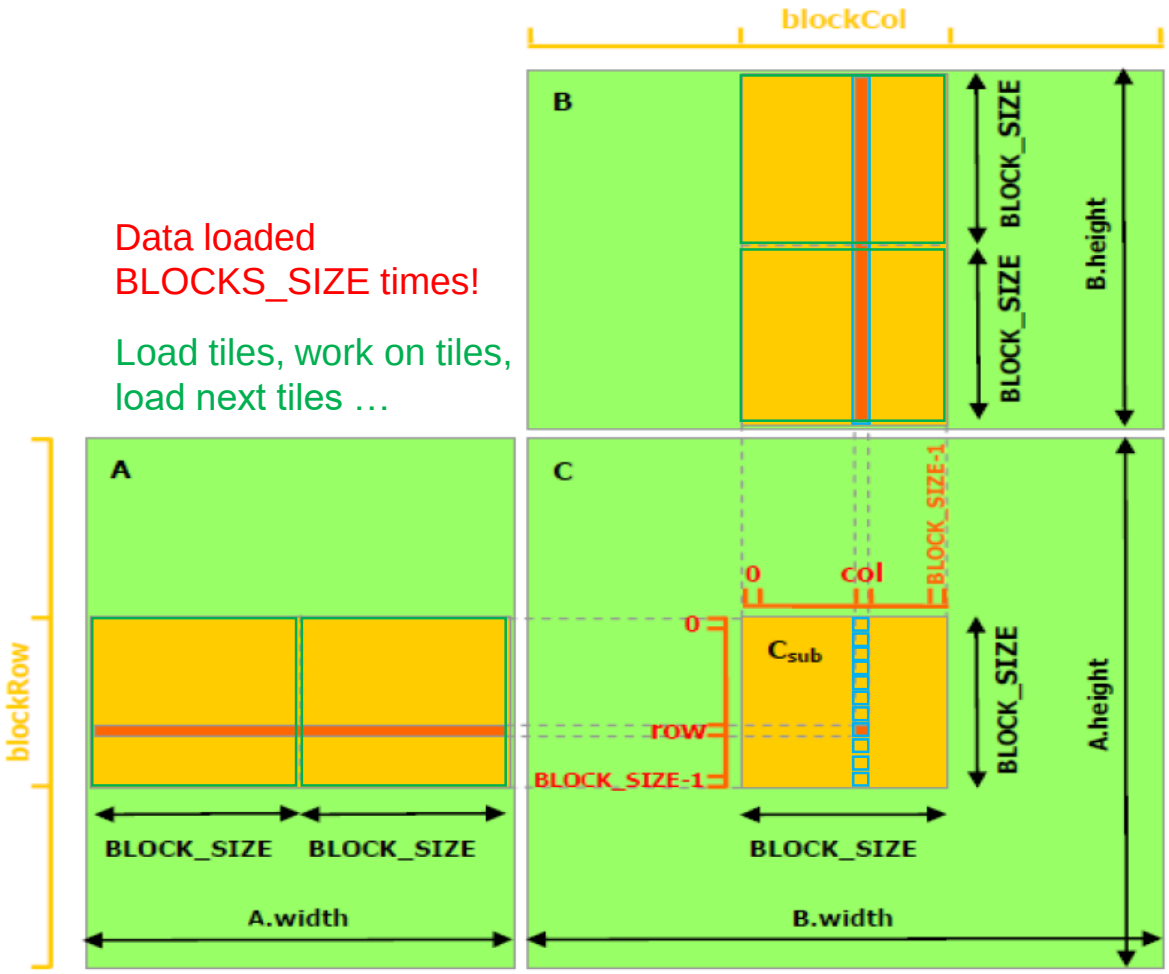
Matrix Multiplication Example



- A lot of memory access with little computations only
- Memory access is all going to slow global device memory
- In a block the same memory is needed by multiple threads

→use shared memory to load one tile of data, consume the data together, advance to next block

Matrix Multiplication Example



Matrix Multiplication Example

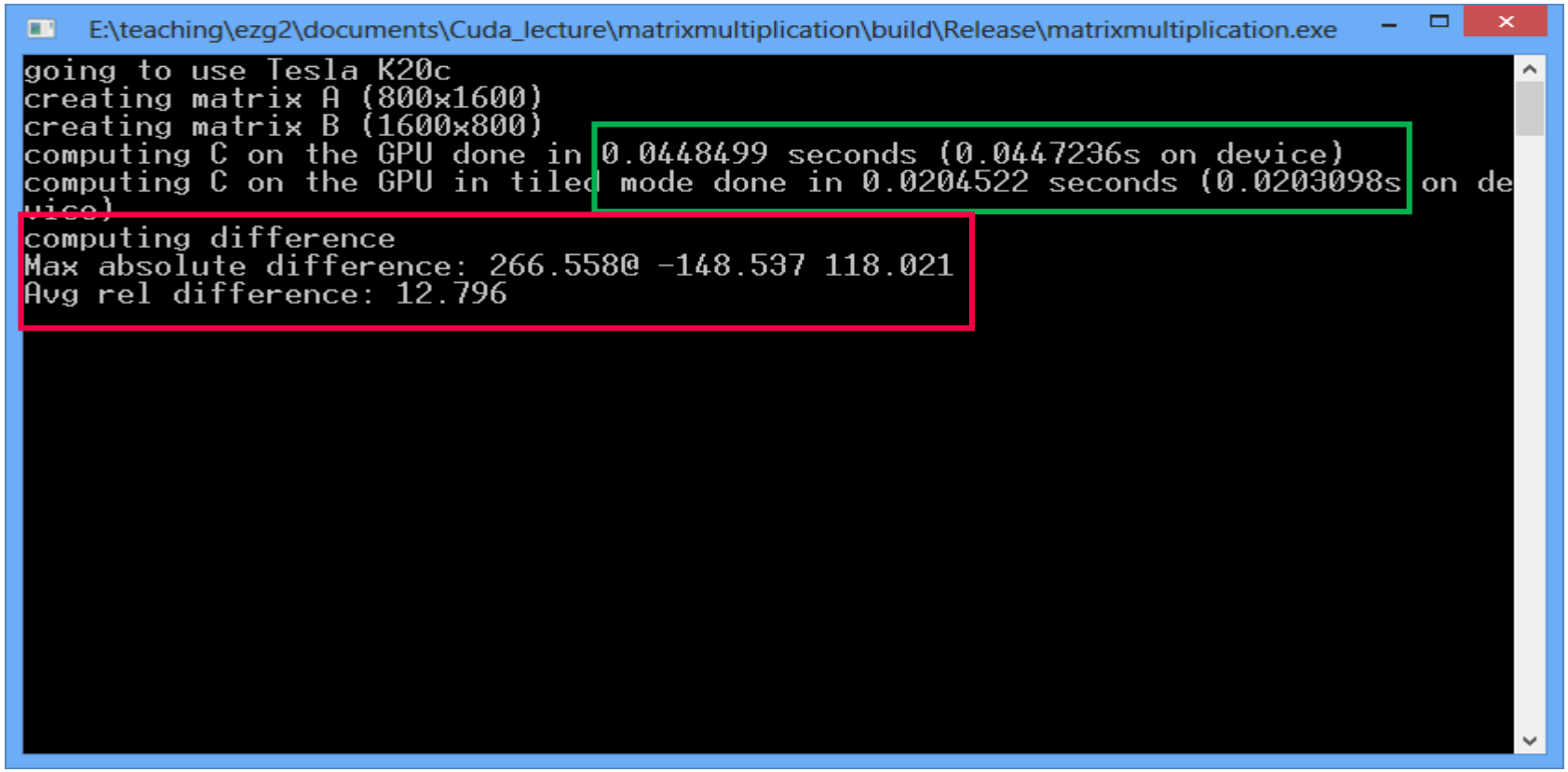


```
68  __global__ void d_multiply_tiled(const float* A, int A_cols, int A_rows,
69                                   const float* B, int B_cols, int B_rows,
70                                   float* C)
71  {
72    __shared__ float s_A_TILE[TILE_WIDTH][TILE_WIDTH];
73    __shared__ float s_B_TILE[TILE_WIDTH][TILE_WIDTH];
74
75    int x = blockIdx.x*TILE_WIDTH + threadIdx.x;
76    int y = blockIdx.y*TILE_WIDTH + threadIdx.y;
77
78    float c_xy = 0;
79    int elements = A_cols;
80    int tiles = elements/TILE_WIDTH;
81    //loop over tiles
82    for(int tile = 0; tile < tiles; ++tile)
83    {
84        //load to shared
85        s_A_TILE[threadIdx.y][threadIdx.x] = A[y*A_cols + tile*TILE_WIDTH + threadIdx.x];
86        s_B_TILE[threadIdx.y][threadIdx.x] = B[(tile*TILE_WIDTH + threadIdx.y)*B_cols + x];
87
88        for(int i = 0; i < TILE_WIDTH; ++i)
89          c_xy += s_A_TILE[threadIdx.y][i] * s_B_TILE[i][threadIdx.x];
90
91    }
92    C[y*B_cols + x] = c_xy;
93  }
```

Blocksize: TILE_WIDTH x TILE_WIDTH

```
c_xy += A[y*A_cols + i]*B[i*B_cols + x];
```

Matrix multiplication problems



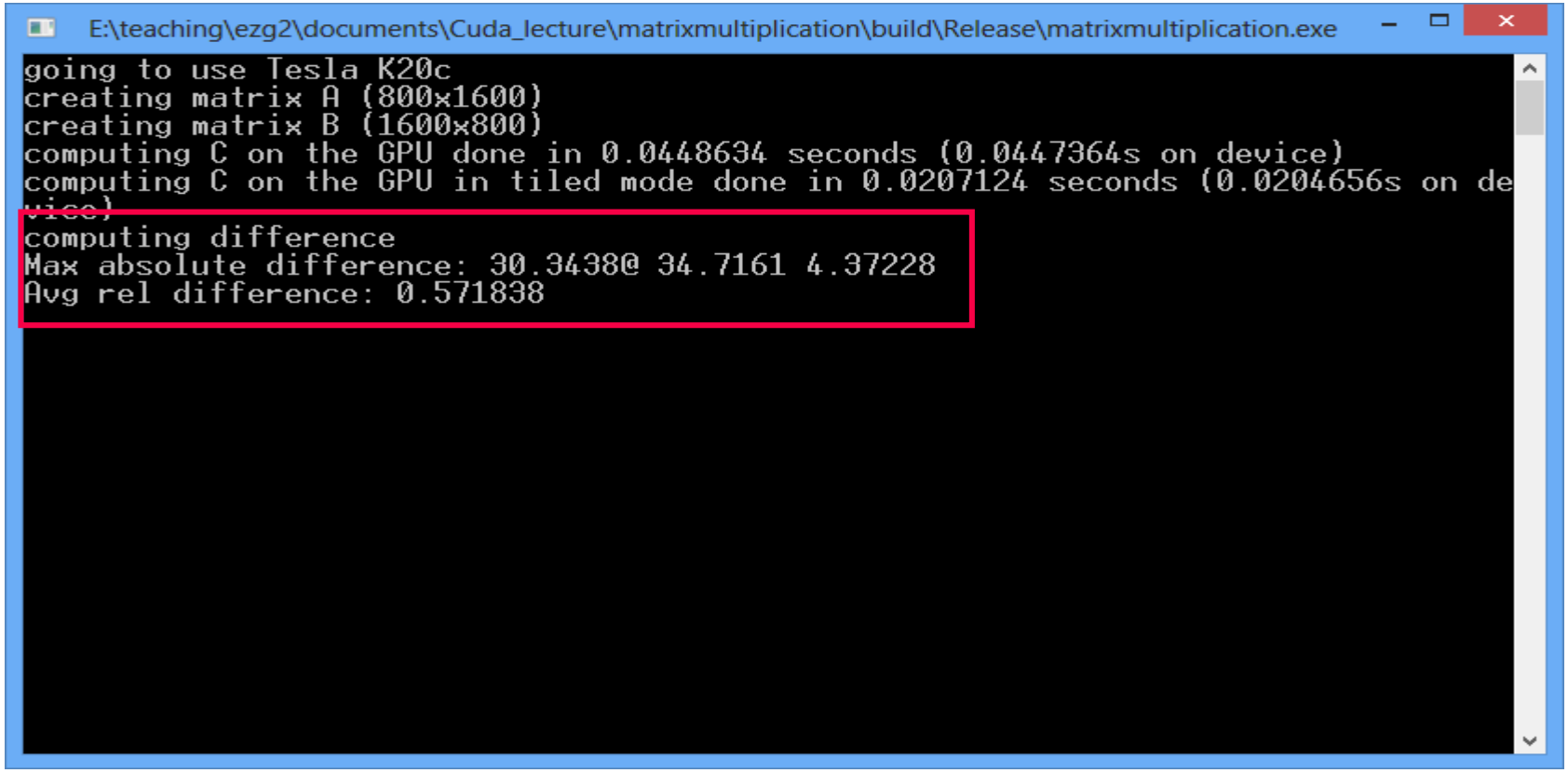
Matrix multiplication problems



```
68  __global__ void d_multiply_tiled(const float* A, int A_cols, int A_rows,
69                                   const float* B, int B_cols, int B_rows,
70                                   float* C)
71  {
72    __shared__ float s_A_TILE[TILE_WIDTH][TILE_WIDTH];
73    __shared__ float s_B_TILE[TILE_WIDTH][TILE_WIDTH];
74
75    int x = blockIdx.x*TILE_WIDTH + threadIdx.x;
76    int y = blockIdx.y*TILE_WIDTH + threadIdx.y;
77
78    float c_xy = 0;
79    int elements = A_cols;
80    int tiles = elements/TILE_WIDTH;
81    //loop over tiles
82    for(int tile = 0; tile < tiles; ++tile)
83    {
84        //load to shared
85        s_A_TILE[threadIdx.y][threadIdx.x] = A[y*A_cols + tile*TILE_WIDTH + threadIdx.x];
86        s_B_TILE[threadIdx.y][threadIdx.x] = B[(tile*TILE_WIDTH + threadIdx.y)*B_cols + x];
87        __syncthreads();
88        for(int i = 0; i < TILE_WIDTH; ++i)
89          c_xy += s_A_TILE[threadIdx.y][i] * s_B_TILE[i][threadIdx.x];
90
91    }
92    C[y*B_cols + x] = c_xy;
93  }
```

Read from another thread before loaded!!

Matrix multiplication problems

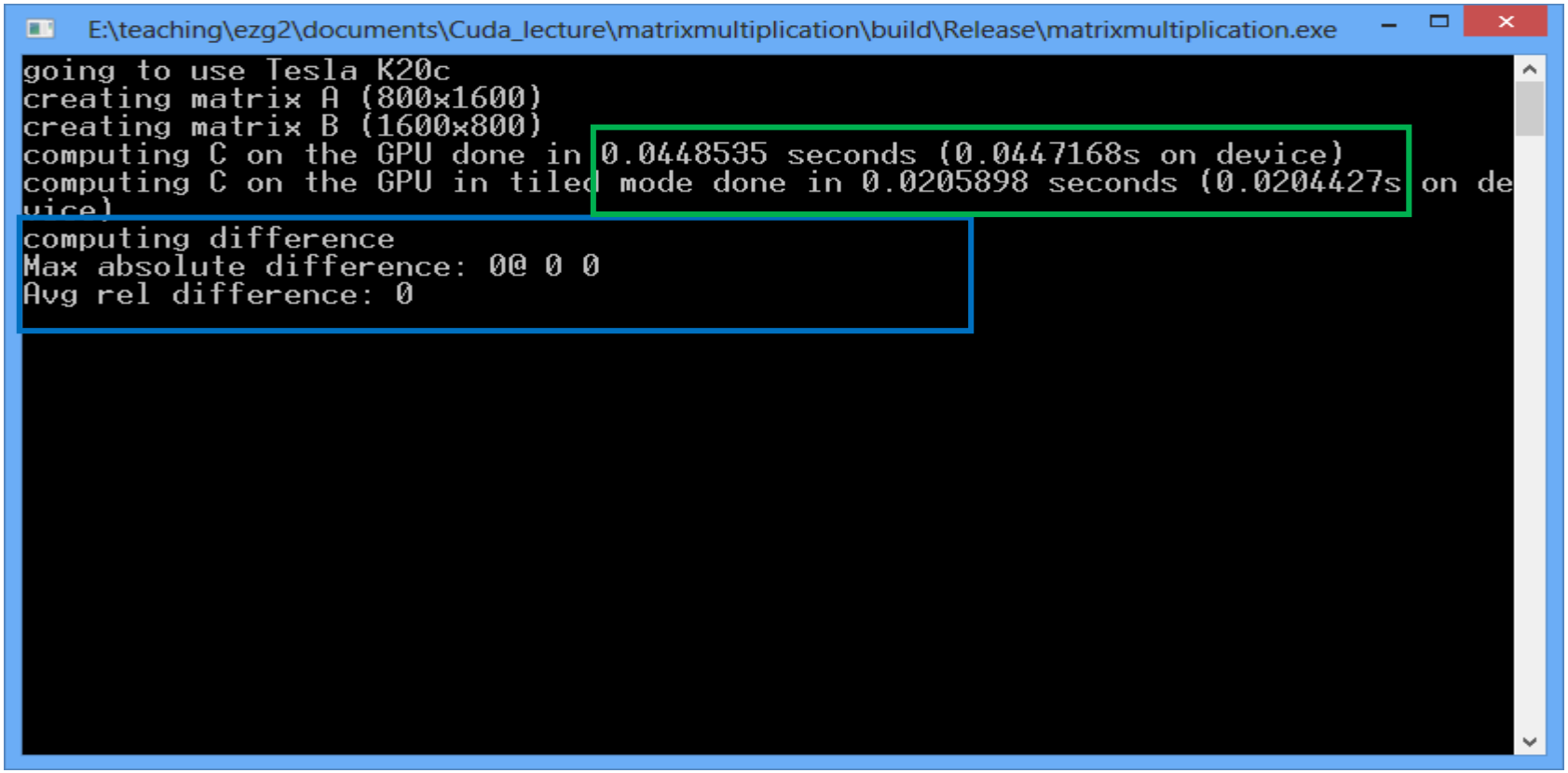


Matrix multiplication problems

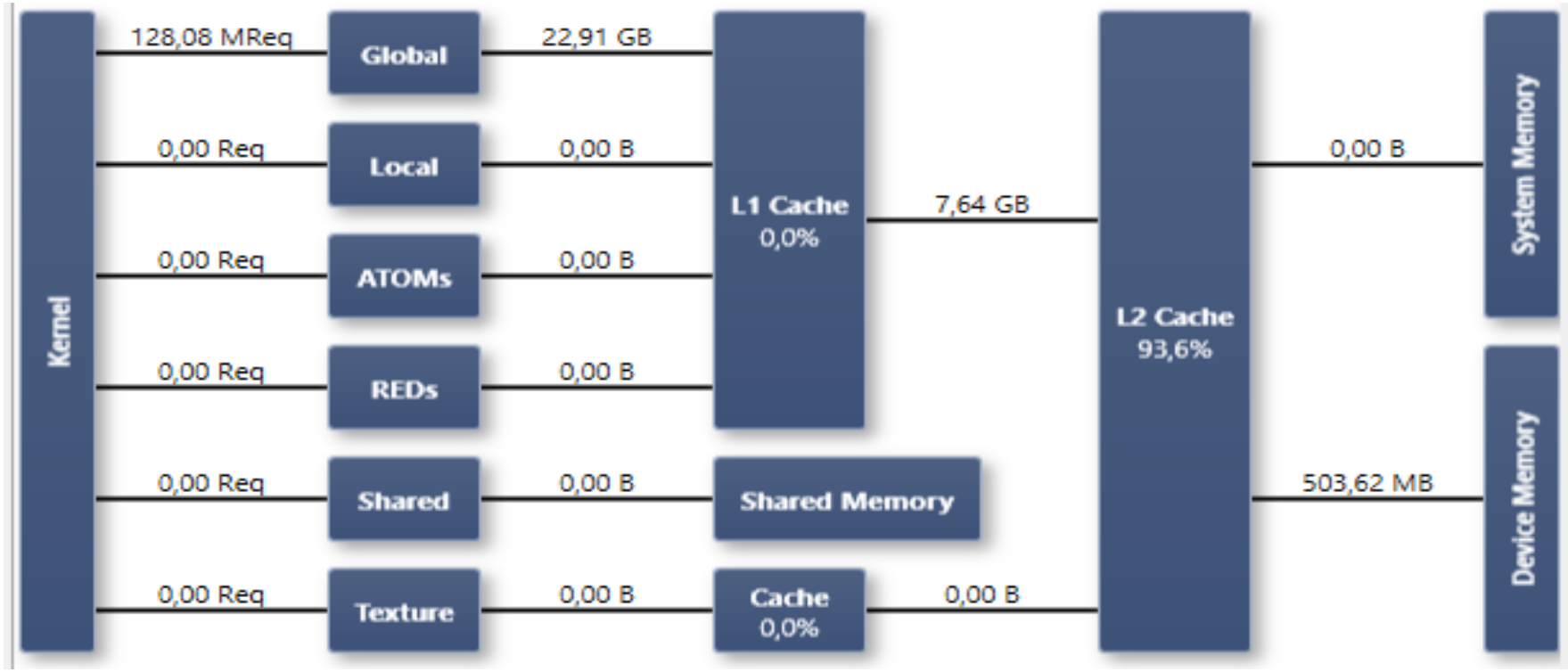


```
68  __global__ void d_multiply_tiled(const float* A, int A_cols, int A_rows,
69                                   const float* B, int B_cols, int B_rows,
70                                   float* C)
71  {
72    __shared__ float s_A_TILE[TILE_WIDTH][TILE_WIDTH];
73    __shared__ float s_B_TILE[TILE_WIDTH][TILE_WIDTH];
74
75    int x = blockIdx.x*TILE_WIDTH + threadIdx.x;
76    int y = blockIdx.y*TILE_WIDTH + threadIdx.y;
77
78    float c_xy = 0;
79    int elements = A_cols;
80    int tiles = elements/TILE_WIDTH;
81    //loop over tiles
82    for(int tile = 0; tile < tiles; ++tile)
83    {
84        //load to shared
85        s_A_TILE[threadIdx.y][threadIdx.x] = A[y*A_cols + tile*TILE_WIDTH + threadIdx.x];
86        s_B_TILE[threadIdx.y][threadIdx.x] = B[(tile*TILE_WIDTH + threadIdx.y)*B_cols + x];
87        __syncthreads();
88        for(int i = 0; i < TILE_WIDTH; ++i)
89          c_xy += s_A_TILE[threadIdx.y][i] * s_B_TILE[i][threadIdx.x];
90        __syncthreads();
91    }
92    C[y*B_cols + x] = c_xy;
93  }
```

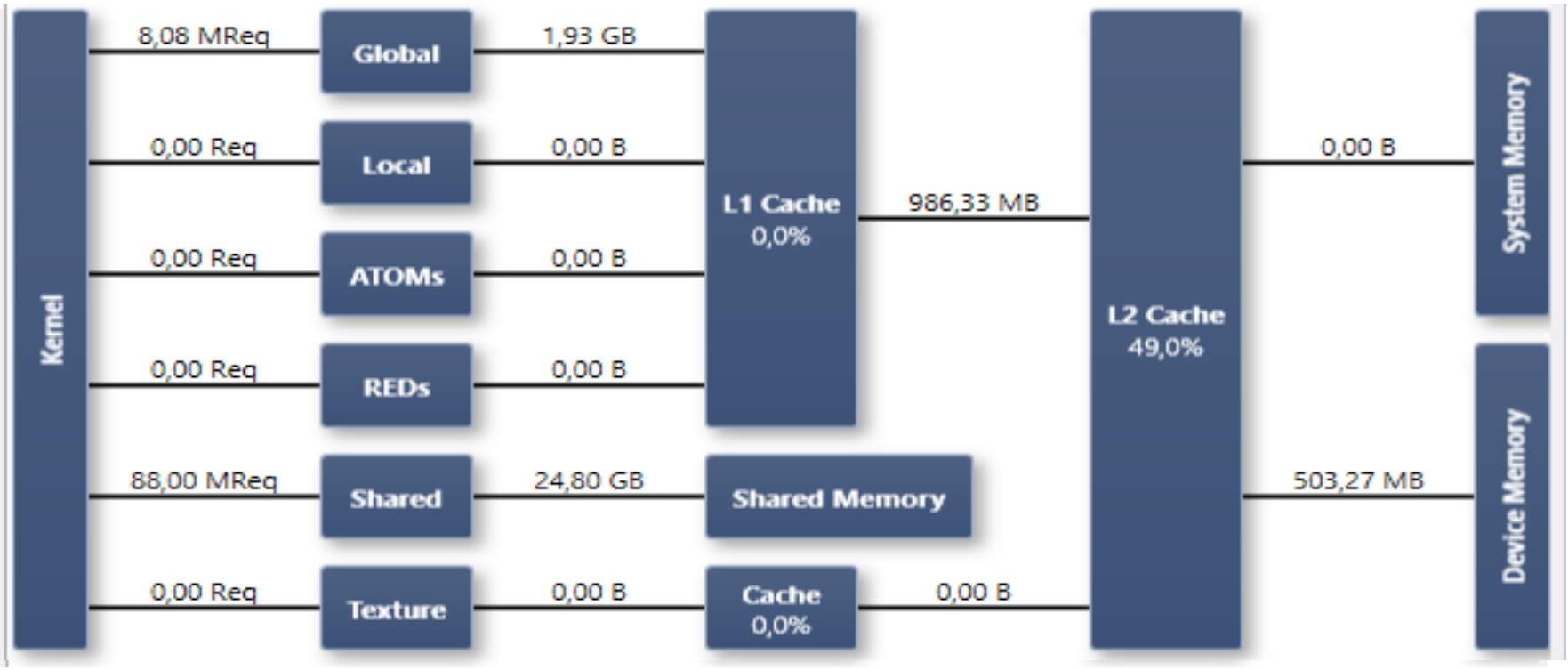
Matrix multiplication problems



Memory statistics: non-tiled

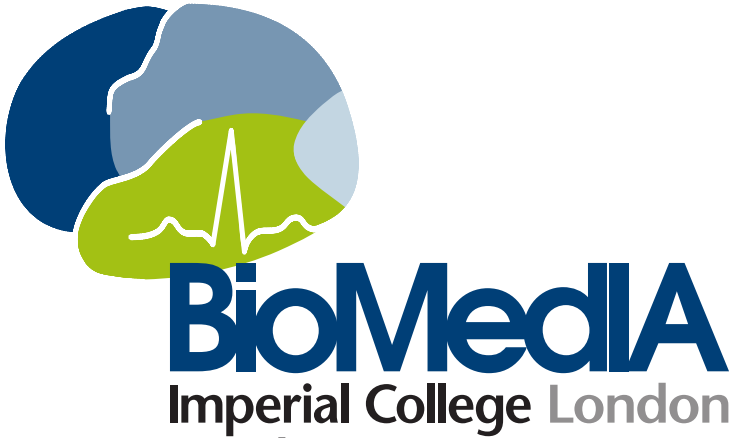


Memory statistics: tiled



Parallel Reduction

Illustrations by Mark Harris, Nvidia



Parallel Reduction

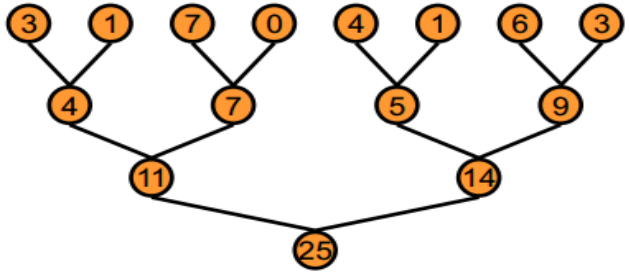


- Common and important data parallel primitive
- Easy to implement in CUDA
  - Harder to get it right
- Serves as a great optimization example
  - Several different versions possible
  - Demonstrates several important optimization strategies

Parallel Reduction



- Tree-based approach used within each thread block



- Need to be able to use multiple thread blocks
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array
- Communicate partial results between thread blocks?

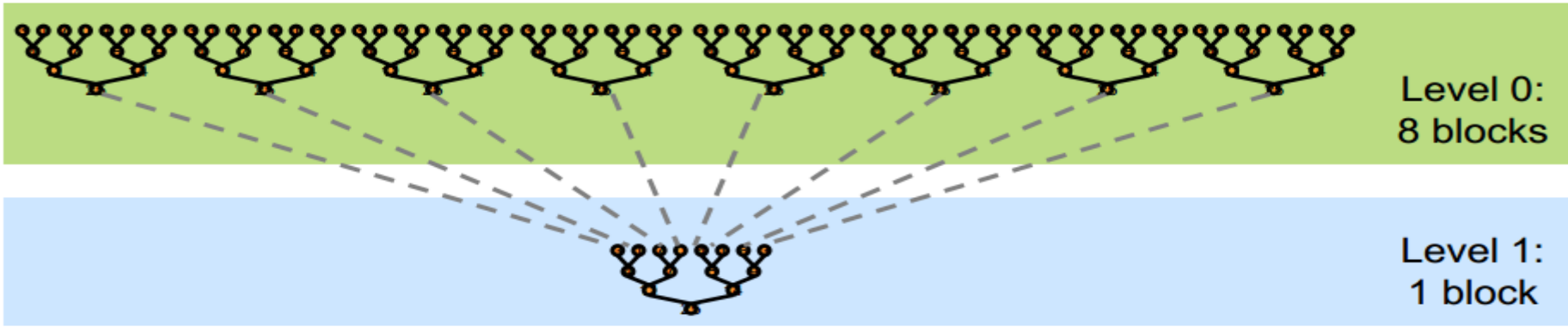
Parallel Reduction



- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively
- But CUDA has no global synchronization. Why?
  - Expensive to build in hardware for GPUs with high processor count
  - Would force programmer to run fewer blocks (no more than # multiprocessors * # resident blocks / multiprocessor) to avoid deadlock, which may reduce overall efficiency
- **Solution:** decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

Parallel Reduction

- Avoid global sync by decomposing computation into multiple kernel invocations



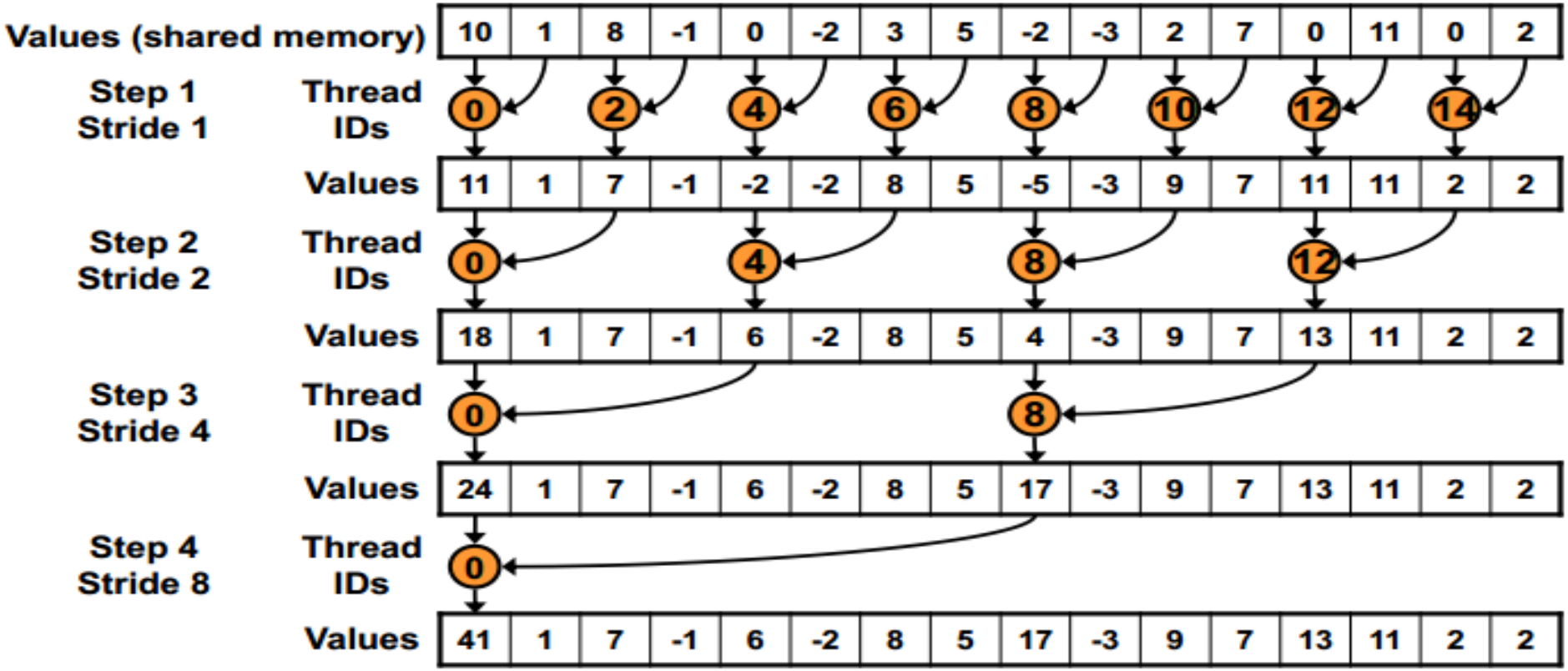
- In the case of reductions, code for all levels is the same
  - Recursive kernel invocation

Parallel Reduction



- **Your turn**:
  - Take two memory elements. Add them together, put one in your pocket.
  - Take the memory element from your neighbour on your left on my command.
  - Add the two numbers (1) together and write down result in “Step 1”, put the other memory element in your pocket.
  - Take the memory element from your the next neighbour on your left who has still a memory element.
  - Add the two numbers together and write down result in “Step 2”, other one in pocket.
  - Continue on my command until only he most right column has memory elements.
  - Pass down the column to the next neighbour with a memory element and add numbers together, write in next empty Step “field”, put other element away.
  - Continue until only one thread (student) is left = result.

Parallel Reduction – Interleaved Addressing



Parallel Reduction



- We should strive to reach GPU peak performance
- Choose the right metric:
  - GFLOP/s: for compute-bound kernels
  - Bandwidth: for memory-bound kernels
- Reductions have very low arithmetic intensity
  - 1 flop per element loaded (bandwidth-optimal)
- Therefore we should strive for peak bandwidth
- Will use G80 for this example
  - 384-bit memory interface, 900 MHz DDR
  - 384 * 1800 / 8 = 86.4 GB/s

Parallel Reduction

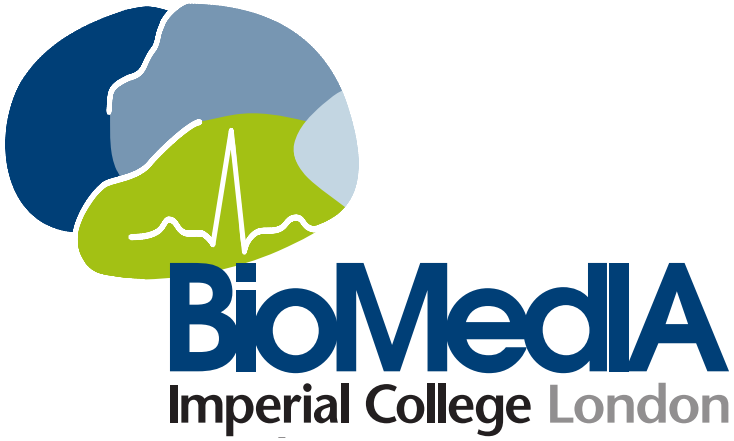


- Many optimizations possible, the one we did is the least efficient
- Very good discussion: https://people.maths.ox.ac.uk/gilesm/cuda/prac4/reduction.pdf

	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

On G80 architecture

Examples and applications



Real-time optical flow



- https://www.youtube.com/watch?v=1D93RmW_eN4

Real time medical image analysis and visualization



https://www.youtube.com/watch?v=mHO6gCm9EP4

KinectFusion



- Developed at DOC@Imperial
- https://www.youtube.com/watch?v=quGhaggn3cQ

KinectFusion

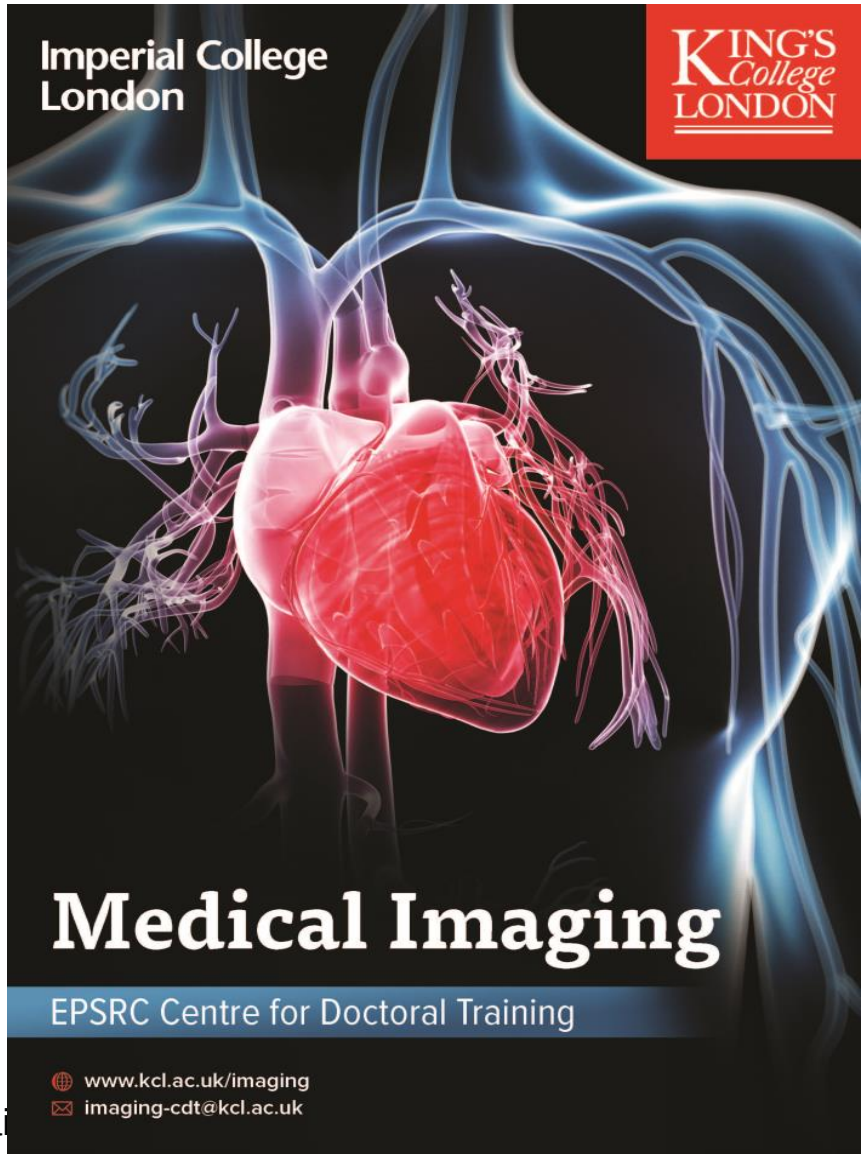


- https://www.youtube.com/watch?v=fE_FnG4RAm8

Imperial College London

Medical Imaging
EPSRC Centre for Doctoral Training

- **15 fully-funded PhD places** for graduates in physics, chemistry, biology, engineering and mathematics
  - Integrated cross-institutional training programme (MRes + 3-year PhD)
- **Research projects** available for:
  - Image acquisition & reconstruction
  - Imaging chemistry & biology
  - Image computing & computational modelling
- **Apply by 4 January 2016:**
  - www.imagingcdt.com
  - Contact: imaging-cdt@kcl.ac.uk



GPU programming

Dr. Bernhard Kainz

