

Big Data Analytics

A. Parallel Computing / 3. Graphical Processing Units (GPUs)

Lars Schmidt-Thieme

Information Systems and Machine Learning Lab (ISMLL)
Institute for Computer Science
University of Hildesheim, Germany

Syllabus

- | | | |
|------------|------|---|
| Tue. 10.4. | (1) | 0. Introduction |
| | | A. Parallel Computing |
| Tue. 17.4. | (2) | A.1 Threads |
| Tue. 24.4. | (3) | A.2 Message Passing Interface (MPI) |
| Tue. 1.5. | — | — <i>Labour Day</i> — |
| Tue. 8.5. | (4) | A.3 Graphical Processing Units (GPUs) |
| Tue. 15.5. | (5) | (ctd.) |
| Tue. 22.5. | — | — <i>Pentecoste Break</i> — |
| | | B. Distributed Storage |
| Tue. 29.5. | (6) | B.1 Distributed File Systems |
| Tue. 5.6. | (7) | B.2 Partitioning of Relational Databases |
| Tue. 12.6. | (8) | B.3 NoSQL Databases |
| | | C. Distributed Computing Environments |
| Tue. 19.6. | (9) | C.1 Map-Reduce |
| Tue. 26.6. | (10) | C.2 Resilient Distributed Datasets (Spark) |
| Tue. 3.7. | (11) | C.3 Computational Graphs (TensorFlow) |
| | | D. Distributed Machine Learning Algorithms |
| Tue. 10.7. | (12) | D.1 Distributed Stochastic Gradient Descent |

Outline

1. GPUs vs CPUs
2. Basics of GPU Programming
3. Example: Color to Grayscale
4. Example: Matrix Multiplication
5. Block Shared Memory

Outline

1. GPUs vs CPUs
2. Basics of GPU Programming
3. Example: Color to Grayscale
4. Example: Matrix Multiplication
5. Block Shared Memory

Massively Parallel Computation

- ▶ **mildly parallel computation:**
 - ▶ tens of cores of a CPU
- ▶ **massively parallel computation:**
 - ▶ thousands of cores
 - ▶ examples:
 - ▶ compute cluster
 - ▶ computing / data center
 - ▶ grid computing
 - ▶ graphical processing units (GPUs)

Flynn's Taxonomy of Computer Architectures

		instructions	
		single	multiple
data	single	SISD: <ul style="list-style-type: none"> ▶ old single-core CPUs 	MISD: <ul style="list-style-type: none"> ▶ unusual
	multiple	SIMD: <ul style="list-style-type: none"> ▶ GPUs ▶ SIMD operations in ordinary CPUs 	MIMD: <ul style="list-style-type: none"> ▶ multi-core CPUs ▶ multiple CPUs

- ▶ Terminology goes back to Flynn [1972].

Streaming Multiprocessors (SMs)

- ▶ multiple cores executing the same instruction in parallel on different data
 - ▶ **instruction level parallelism**
- ▶ shared resources:
 - ▶ shared memory
- ▶ much simpler than a full-fledged CPU:
 - ▶ slower clockrate
 - ▶ less cache
 - ▶ no branch prediction, no speculative execution

NVIDIA GPU Generations / Streaming Multiprocessors

architecture	release	cores /SM	tensor- cores/SM	shared mem. /SM [kB]	comp. cap.
Volta	2017	64	8	64–96	7.0
Pascal	2016	64	—	64–96	6.x
Maxwell	2014	128	—	96	5.x
Kepler	2012	192	—	16–48	3.x
Fermi	2010	32	—	64	2.x
Tesla	2006	8	—		1.x

“compute capability” describes available features for the GPU.

see <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities>

Note: Some pascal GPUs have 128 cores/SM.

NVIDIA GPUs

name	generation	GPUs	SMs	cores/SM	cores	comp. cap.
Tesla V100	Volta	1	80	64	5120	7.0
Tesla P100	Pascal	1	56	64	3584	6.0
Tesla M60	Maxwell	1	24	128	3072	5.2
Tesla K80	Kepler	2	13	192	4992	3.7
Quadro GV100	Volta	1	80	64	5120	7.0
Quadro P6000	Pascal	1	60	64	3840	6.1
Quadro M6000	Maxwell	1	24	128	3072	5.2
Quadro K6000	Kepler	1	15	192	2880	3.5
GeForce TITAN Xp	Pascal	1	60	64	3840	6.1
GeForce TITAN X	Maxwell	1	24	128	3072	6.1
GeForce TITAN	Kepler	1	14	192	2688	3.5
GeForce GTX 1080 Ti	Pascal	1	56	64	3584	6.1
GeForce GTX 980 Ti	Maxwell	1	22	128	2816	5.2
GeForce GTX 780 Ti	Kepler	1	15	192	2880	3.5
Quadro K2100M	Kepler	1	3	192	576	3.0

Peak Performance

name	generation	cores	clock speed [MHz]	peak performance [FLOPS]
Tesla V100	Volta	5376	1200-1455	15.7 T (tensor: 125 T)
Tesla P100	Pascal	3584	1328-1480	10.6 T
Tesla M60	Maxwell	3072	948-1114	6.8 T
Tesla K40	Kepler	2496	745- 875	4.3 T
Intel Core i7-7920HQ	Kaby Lake-H	4	3174-4198	0.033 T

- ▶ to compute peak performance:

$$\text{peakPerformance} = 2 \cdot \text{numberOfCores} \cdot \text{clockSpeed}$$

- ▶ where does the factor 2 come from?
 - ▶ Modern GPUs and CPUs have fused-multiply-add instructions (FMA), where one instruction basically performs 2 operations:

$$d := \text{round}(a \cdot b + c)$$

- ▶ since Intels Haswell architecture in 2013.

Outline

1. GPUs vs CPUs
2. Basics of GPU Programming
3. Example: Color to Grayscale
4. Example: Matrix Multiplication
5. Block Shared Memory

GPU Programming

- ▶ **Compute Unified Device Architecture (CUDA):**
 - ▶ C/C++ language extension
 - ▶ dedicated preprocessor: `nvcc`
 - ▶ development tools, e.g., profiler, debugger
 - ▶ runtime API
 - ▶ for C, C++, Fortran
 - ▶ proprietary by Nvidia
 - ▶ PyCUDA: language binding for Python
- ▶ **Open Computing Language (OpenCL):**
 - ▶ interface for parallel computing across heterogeneous hardware
 - ▶ including GPUs
 - ▶ C/C++ like language
 - ▶ open standard managed by Khronos Compute Working Group
 - ▶ Apple, IBM, AMD, Intel, Qualcomm, Nvidia
 - ▶ PyOpenCL: a language binding for Python

GPU program abstraction

- ▶ execute a procedure (**kernel**) over a cartesian product / **grid** of 1 to 3 integer ranges
 - ▶ ranges are called x, y, z .
 - ▶ each range starts at 0.
 - ▶ example: all (x,y) pixel coordinates of an image.
- ▶ elements are grouped into **blocks** / tiles of the grid
 - ▶ fixed block size for each range x, y, z
- ▶ elements are usually coordinates / indices of some data.
 - ▶ used to compute memory address.
 - ▶ used to make control decisions.

GPU program abstraction

- ▶ elements are loaded into GPU registers and accessible through symbolic names in kernels:
 - ▶ number of blocks: **gridDim**
 - ▶ block size: **blockDim**
 - ▶ block index: **blockIdx**
 - ▶ relative index in the block: **threadIdx**
 - ▶ each variable is of type **dim3**, having components x , y and z .
 - ▶ element can be computed via:

$$\text{elem.x} := \text{blockDim.x} \cdot \text{blockIdx.x} + \text{threadIdx.x}$$

$$\text{elem.y} := \text{blockDim.y} \cdot \text{blockIdx.y} + \text{threadIdx.y}$$

$$\text{elem.z} := \text{blockDim.z} \cdot \text{blockIdx.z} + \text{threadIdx.z}$$

- ▶ the total grid size is

$$\text{size.x} := \text{blockDim.x} \cdot \text{gridDim.x}$$

$$\text{size.y} := \text{blockDim.y} \cdot \text{gridDim.y}$$

$$\text{size.z} := \text{blockDim.z} \cdot \text{gridDim.z}$$

CPU/GPU Cooperation

1. allocate unified memory on GPU/CPU

```
1      cudaMallocManaged(&ptr, size);
```

2. start GPU computation

```
1      kernel<<<numberOfBlocks, sizeOfBlocks>>>(kernelParams...);
```

3. wait for GPU program to complete

```
1      cudaDeviceSynchronize();
```

4. free memory

```
1      cudaFree(ptr);
```

Kernel Calls

code:

```
1 kernel<<<numberOfBlocks, sizeofBlocks>>>(kernelParams...);
```

executes on GPU as (for one-dimensional indices):

```
1 gridDim.x = numberOfBlocks; blockDim.x = sizeofBlocks;
2 for blockIdx.x = 0 ... numberOfBlocks.x-1 in parallel:
3   for threadIdx.x = 0 ... sizeofBlocks.x-1 in parallel:
4     kernel(kernelParams ...);
```

executes on GPU as (for three-dimensional indices):

```
1 gridDim = numberOfBlocks; blockDim = sizeofBlocks;
2 for blockIdx.x = 0 ... numberOfBlocks.x-1 in parallel:
3   for blockIdx.y = 0 ... numberOfBlocks.y-1 in parallel:
4     for blockIdx.z = 0 ... numberOfBlocks.z-1 in parallel:
5       for threadIdx.x = 0 ... sizeofBlocks.x-1 in parallel:
6         for threadIdx.y = 0 ... sizeofBlocks.y-1 in parallel:
7           for threadIdx.z = 0 ... sizeofBlocks.z-1 in parallel:
8             kernel(kernelParams ...);
```


Example: Add Two Vectors

```

1 #include <iostream>
2 #include <math.h>
3 #define CEIL(x,y) (x+y-1)/y
4
5 __global__ void add(float *x, float *y) {
6     int n = blockDim.x * blockDim.x + threadIdx.x;
7     y[n] = x[n] + y[n];
8 }
9
10 int main(void) {
11     int N = 1<<20;
12     float *x, *y;
13     cudaMallocManaged(&x, N*sizeof(float));
14     cudaMallocManaged(&y, N*sizeof(float));
15     for (int n = 0; n < N; n++) {
16         x[n] = 1.0f;
17         y[n] = 2.0f;
18     }
19
20     add<<<CEIL(N, 256), 256>>>(x, y);
21     cudaDeviceSynchronize();
22
23     float maxError = 0.0f;
24     for (int n = 0; n < N; n++)
25         maxError = fmax(maxError, fabs(y[n]-3.0f));
26     std::cout << "Max error: " << maxError << std::endl;
27
28     cudaFree(x);
29     cudaFree(y);
30     return 0;
31 }

```

Compiling CUDA Code

- ▶ code in file `ex-add.cu`.
- ▶ code is run through a preprocessor `nvcc`

```
1  nvcc -o ex-add ex-add.cu
```

- ▶ `nvcc` separates host code and device code (e.g., kernels),
- ▶ compiles host code by default `c++` compiler, e.g., `gcc`.
 - ▶ kernel calls are substituted by code that
 - loads the device code into the GPU and
 - starts the computation.
- ▶ compiles device code by device code compiler.

If Overall Range is not a Multiple of the Blocksize

- ▶ then the **range has to be extended** to the next multiple.
 - ▶ use the ceiling of the $\text{range_size}/\text{block_size}$ ratio.
- ▶ If the range is extended, the kernel needs to be **guarded** not to access out-of-index locations.
 - ▶ the correct sizes have to be passed as parameters.

Example: Add Two Vectors (Guarded)

```

1 #include <iostream>
2 #include <math.h>
3 #define CEIL(x,y) (x+y-1)/y
4
5 __global__ void add(float *x, float *y, int N) {
6     int n = blockIdx.x * blockDim.x + threadIdx.x;
7     if (n < N)
8         y[n] = x[n] + y[n];
9 }
10
11 int main(void) {
12     int N = 1<<20;
13     float *x, *y;
14     cudaMallocManaged(&x, N*sizeof(float));
15     cudaMallocManaged(&y, N*sizeof(float));
16     for (int n = 0; n < N; n++) {
17         x[n] = 1.0f;
18         y[n] = 2.0f;
19     }
20
21     add<<<CEIL(N, 198), 198>>>(x, y, N);
22     cudaDeviceSynchronize();
23
24     float maxError = 0.0f;
25     for (int n = 0; n < N; n++)
26         maxError = fmax(maxError, fabs(y[n]-3.0f));
27     std::cout << "Max error: " << maxError << std::endl;
28
29     cudaFree(x);
30     cudaFree(y);
31     return 0;

```

Outline

1. GPUs vs CPUs
2. Basics of GPU Programming
3. Example: Color to Grayscale
4. Example: Matrix Multiplication
5. Block Shared Memory

Unmanaged GPU Memory

- ▶ If existing data structures on the host should be used, unified/managed CPU/GPU memory usually cannot be used.
 - ▶ e.g., openCV matrix class **Mat**.

- ▶ more finegrained control:

1a. allocate GPU memory

```
1 cudaMalloc(&d_ptr, size);
```

1b. transfer input data from main memory to GPU memory

```
1 cudaMemcpy(d_ptr, h_ptr, size, cudaMemcpyHostToDevice);
```

2. start GPU computation

3. wait for GPU program to complete

4a. transfer output data from GPU memory to main memory

```
1 cudaMemcpy(h_ptr, d_ptr, size, cudaMemcpyDeviceToHost);
```

4b. free GPU memory

```
1 cudaFree(d_ptr);
```

Note: `d_ptr` is a pointer to device memory, `h_ptr` to host memory.

Example: Color to Grayscale

- ▶ a **color image** is stored with 3 channels
 - ▶ one each for a red, green, and blue component (RGB)
 - ▶ each channel stores an intensity
 - ▶ e.g, as an unsigned char 0, . . . , 255.
- ▶ a **grayscale image** is stored with a single channel
 - ▶ storing a grayscale intensity
 - ▶ e.g, as an unsigned char 0, . . . , 255.
- ▶ to convert a color image to a grayscale image, average the color channel intensities:

$$\text{intensityGray} := \frac{\text{intensityRed} + \text{intensityGreen} + \text{intensityBlue}}{3}$$

- ▶ an image with R rows, C columns and H channels is stored as $R \times C \times H$ array
 - ▶ element (r, c, h) is at linear index

$$\text{index}(r, c, h) := (r \cdot C + c) \cdot H + h$$

Example: Color to Grayscale

```

1 #include <opencv2/highgui/highgui.hpp>
2
3 #define DTYPE unsigned char
4
5 __global__ void rgb2gray(const DTYPE* in, DTYPE* out) {
6     int index_out = blockIdx.x * blockDim.x + threadIdx.x;
7     int index_in = index_out * 3;
8     out[index_out] = (in[index_in] + in[index_in + 1] + in[index_in + 2]) / 3;
9 }
10
11 int main(int arg, char* args []) {
12     cv::Mat img = cv::imread("dom_hildesheim.png", CV_LOAD_IMAGE_COLOR);
13     cv::Mat img_gray (img.rows, img.cols, CV_8UC1);
14
15     int img_size = img.rows * img.cols * img.channels() * sizeof(DTYPE),
16         img_gray_size = img_gray.rows * img_gray.cols * img_gray.channels() * sizeof(DTYPE);
17     DTYPE *in = 0, *out = 0;
18     cudaMalloc(&in, img_size);
19     cudaMalloc(&out, img_gray_size);
20     cudaMemcpy(in, img.data, img_size, cudaMemcpyHostToDevice);
21
22     rgb2gray<<<img.rows, img.cols>>>(in, out);
23     cudaDeviceSynchronize ();
24
25     cudaMemcpy(img_gray.data, out, img_gray_size, cudaMemcpyDeviceToHost);
26     cudaFree(in);
27     cudaFree(out);
28     cv::imwrite("out-gray.png", img_gray);
29 }
  
```


Example: Color to Grayscale

input image:



output image:



Example: Color to Grayscale (alternative grid)

```

1 #include <opencv2/highgui/highgui.hpp>
2
3 #define DTYPE unsigned char
4 #define CEIL(A,B) (A+B-1)/B
5
6 __global__ void rgb2gray(const DTYPE* in, DTYPE* out, int num_rows, int num_cols) {
7     int row = blockIdx.x * blockDim.x + threadIdx.x,
8         col = blockIdx.y * blockDim.y + threadIdx.y;
9     if (row < num_rows && col < num_cols) {
10        int index_out = row * num_cols + col;
11        int index_in = index_out * 3;
12        out[index_out] = (in[index_in] + in[index_in + 1] + in[index_in + 2]) / 3;
13    }
14 }
15
16 int main(int arg, char* args []) {
17     cv::Mat img = cv::imread("dom_hildesheim.png", CV_LOAD_IMAGE_COLOR);
18     cv::Mat img_gray (img.rows, img.cols, CV_8UC1);
19
20     int img_size = img.rows * img.cols * img.channels() * sizeof(DTYPE),
21         img_gray_size = img_gray.rows * img_gray.cols * img_gray.channels() * sizeof(DTYPE);
22     DTYPE *in = 0, *out = 0;
23     cudaMalloc(&in, img_size);
24     cudaMalloc(&out, img_gray_size);
25     cudaMemcpy(in, img.data, img_size, cudaMemcpyHostToDevice);
26
27     rgb2gray<<<dim3(CEIL(img.rows,32), CEIL(img.cols,32)), dim3(32,32)>>>(in, out, img.rows, img.cols);
28     cudaDeviceSynchronize();
29
30     cudaMemcpy(img_gray.data, out, img_gray_size, cudaMemcpyDeviceToHost);
31     cudaFree(in);

```

Block Hardware Limitations

- ▶ maximum # threads / block: 1024
 - ▶ e.g., 32×32 patches for images (or tiles for matrices).
 - ▶ first color2gray example works only for images with maximal 1024 columns!
second color2gray example works always.
- ▶ maximum # threads / SM: 2048
 - ▶ e.g., 2 full blocks a 1024 threads.
- ▶ maximum # blocks / SM: 32 (Maxwell,Pascal,Volta; 16 for Kepler)

What are Blocks Good For?

- ▶ all elements of a block are executed on the same SM
- ▶ each block is executed in scheduling units of 32 elements (**warps**)
 - ▶ all threads in a warp execute the same instruction (“in lockstep”)
 - ▶ zero-overhead warp scheduling:
 - ▶ eligible: operands for next operation is ready
 - ▶ scheduling selects from eligible warps based on prioritization
- ▶ if instructions of threads within a warp diverge
e.g., because of a diverging **if** statement,
then the warp is split in subgroups which are executed sequentially.
- ▶ Thus, avoid diverging control flows where possible.
- ▶ Threads of the same block can share memory (see two sections below).

Outline

1. GPUs vs CPUs
2. Basics of GPU Programming
3. Example: Color to Grayscale
- 4. Example: Matrix Multiplication**
5. Block Shared Memory

Example: Matrix Multiplication (CPU, direct)

```

1 #include <stdlib.h>
2
3 class Matrix {
4 public:
5     int _N, _M;
6     float* _data;
7
8     Matrix(int N, int M)
9         : _N(N), _M(M), _data(new float[N*M]) {
10         for (int n = 0; n < N; ++n)
11             for (int m = 0; m < M; ++m)
12                 _data[n*_M + m] = rand() * 2.0f
13                     / RAND_MAX - 1.0f;
14     }
15
16     float& operator()(int n, int m) {
17         return _data[n*_M + m];
18     }
19 };

```

```

1 #include "Matrix.h"
2
3 void mult(Matrix& A, Matrix& B, Matrix& C) {
4     const int N = A._N, M = A._M, L = B._M;
5     for (int n = 0; n < N; ++n) {
6         for (int l = 0; l < L; ++l) {
7             float c = 0;
8             for (int m = 0; m < M; ++m)
9                 c += A(n,m) * B(m,l);
10            C(n,l) = c;
11        }
12    }
13 }
14
15 int main(int argn, char** argv) {
16     const int N = 4096, M = 2048, L = 2048;
17     Matrix A(N,M), B(M,L), C(N,L);
18     mult(A, B, C);
19 }

```

Example: Matrix Multiplication (CPU, tiled)

```

1 #include "Matrix.h"
2 #include <cmath>
3 #include <algorithm>
4
5 void mult(Matrix& A, Matrix& B, Matrix& C) {
6     const int N = A._N, M = A._M, L = B._M, K = ceil(sqrt(M));
7     for (int n = 0; n < N; ++n)
8         for (int l = 0; l < L; ++l)
9             C(n,l) = 0;
10    for (int n0 = 0; n0 < N; n0+= K) {
11        for (int l0 = 0; l0 < L; l0+= K) {
12            for (int m0 = 0; m0 < M; m0+= K) {
13                for (int n = n0; n < std::min(N, n0+K); ++n) {
14                    for (int l = l0; l < std::min(L, l0+K); ++l) {
15                        float c = 0;
16                        for (int m = m0; m < std::min(M, m0+K); ++m)
17                            c += A(n,m) * B(m,l);
18                        C(n,l) += c;
19                    }
20                }
21            }
22        }
23    }
24 }
25
26 int main(int argn, char** argv) {
27     const int N = 4096, M = 2048, L = 2048;
28     Matrix A(N,M), B(M,L), C(N,L);
29     mult(A, B, C);
30 }

```

Example: Matrix Multiplication (GPU, direct)

```

1  #include "Matrix.h"
2
3  __global__ void d_mult(int M, int L, float* A, float* B, float* C) {
4  int n = blockIdx.x * blockDim.x + threadIdx.x;
5  int l = blockIdx.y * blockDim.y + threadIdx.y;
6  float c = 0;
7  for (int m = 0; m < M; ++m)
8      c += A[n*M + m] * B[m*L + l];
9  C[n*L + l] = c;
10 }
11
12 void mult(Matrix& A, Matrix& B, Matrix& C) {
13     const int N = A._N, M = A._M, L = B._M;
14
15     float *d_A, *d_B, *d_C;
16     cudaMalloc(&d_A, N*M*sizeof(float));
17     cudaMalloc(&d_B, M*L*sizeof(float));
18     cudaMalloc(&d_C, N*L*sizeof(float));
19     cudaMemcpy(d_A, A._data, N*M*sizeof(float), cudaMemcpyHostToDevice);
20     cudaMemcpy(d_B, B._data, M*L*sizeof(float), cudaMemcpyHostToDevice);
21
22     dim3 block(16, 16), grid(N/16, L/16);
23     d_mult<<<grid, block>>>(M, L, d_A, d_B, d_C);
24     cudaDeviceSynchronize();
25     cudaMemcpy(C._data, d_C, N*L*sizeof(float), cudaMemcpyDeviceToHost);
26     cudaFree(d_A);
27     cudaFree(d_B);
28     cudaFree(d_C);
29 }
31 int main(int argn, char** argv) {
32     const int N = 4096, M = 2048, L = 2048;
33     Matrix A(N,M), B(M,L), C(N,L);
34     mult(A, B, C);
35 }

```


Outline

1. GPUs vs CPUs
2. Basics of GPU Programming
3. Example: Color to Grayscale
4. Example: Matrix Multiplication
5. Block Shared Memory

Example: Matrix Multiplication (GPU, tiled, v0)

```

1 #include "Matrix.h"
2
3 const int DN = 16, DL = 16, DM = 16;
4
5 __global__ void d_mult(int N, int M, int L,
6                       float * A, float * B, float * C) {
7     int n0 = blockIdx.x, dn = threadIdx.x,
8         l0 = blockIdx.y, dl = threadIdx.y;
9     int n = n0 * DN + dn;
10    int l = l0 * DL + dl;
11    float c = 0;
12    for (int m0 = 0; m0 < M/DM; ++m0)
13        for (int dm = 0; dm < DM; ++dm)
14            c += A[n*M + m0*DM + dm]
15                * B[(m0*DM + dm)*L + l];
16    C[n*L + l] = c;
17 }
18
19 void mult(Matrix& A, Matrix& B, Matrix& C) {
20     const int N = A._N, M = A._M, L = B._M;
21
22     float *d_A, *d_B, *d_C;
23     cudaMalloc(&d_A, N*M*sizeof(float));
24     cudaMalloc(&d_B, M*L*sizeof(float));
25     cudaMalloc(&d_C, N*L*sizeof(float));
26     cudaMemcpy(d_A, A._data, N*M*sizeof(float), cudaMemcpyHostToDevice);
27     cudaMemcpy(d_B, B._data, M*L*sizeof(float), cudaMemcpyHostToDevice);
28
29     dim3 block(DN, DL), grid(N/DN, L/DL);
30     d_mult<<<grid, block>>>(N, M, L, d_A, d_B, d_C);
31     cudaDeviceSynchronize();
32     cudaMemcpy(C._data, d_C, N*L*sizeof(float), cudaMemcpyDeviceToHost);
33     cudaFree(d_A);
34     cudaFree(d_B);
35     cudaFree(d_C);
36 }
37
38
39 int main(int argn, char** argv) {
40     const int N = 4096, M = 2048, L = 2048;
41     Matrix A(N,M), B(M,L), C(N,L);
42     mult(A, B, C);
43 }

```

Block Shared Memory

- ▶ each thread $C(n, l)$ has $2M + 1$ memory accesses.
 - ▶ $A(n, m)$ and $B(m, l)$ for $m = 0, \dots, M - 1$
- ▶ all threads $C(n, l)$ and $C(n, l')$ share M of those
 - ▶ $A(n, m)$ for $m = 0, \dots, M - 1$
- ▶ all threads $C(n, l)$ and $C(n', l)$ share M of those
 - ▶ $B(m, l)$ for $m = 0, \dots, M - 1$

Block Shared Memory

- ▶ each thread $C(n, l)$ has $2M + 1$ memory accesses.
 - ▶ $A(n, m)$ and $B(m, l)$ for $m = 0, \dots, M - 1$
- ▶ all threads $C(n, l)$ and $C(n, l')$ share M of those
 - ▶ $A(n, m)$ for $m = 0, \dots, M - 1$
- ▶ all threads $C(n, l)$ and $C(n', l)$ share M of those
 - ▶ $B(m, l)$ for $m = 0, \dots, M - 1$
- ▶ First idea:
make threads within a block $C(n_0 : n_0 + \Delta N, l_0 : l_0 + \Delta L)$ load tiles $A(n_0 : n_0 + \Delta N, 0 : M - 1)$ and $B(0 : M - 1, l_0 : l_0 + \Delta L)$ into shared memory.
 - ▶ but as shared memory is limited, need to subdivide over m also.

Block Shared Memory

- ▶ each thread $C(n, l)$ has $2M + 1$ memory accesses.
 - ▶ $A(n, m)$ and $B(m, l)$ for $m = 0, \dots, M - 1$
- ▶ all threads $C(n, l)$ and $C(n, l')$ share M of those
 - ▶ $A(n, m)$ for $m = 0, \dots, M - 1$
- ▶ all threads $C(n, l)$ and $C(n', l)$ share M of those
 - ▶ $B(m, l)$ for $m = 0, \dots, M - 1$
- ▶ First idea:

make threads within a block $C(n_0 : n_0 + \Delta N, l_0 : l_0 + \Delta L)$ load tiles $A(n_0 : n_0 + \Delta N, 0 : M - 1)$ and $B(0 : M - 1, l_0 : l_0 + \Delta L)$ into shared memory.

 - ▶ but as shared memory is limited, need to subdivide over m also.
- ▶ Second idea:

make threads within a block $C(n_0 : n_0 + \Delta N, l_0 : l_0 + \Delta L)$ load tiles $A(n_0 : n_0 + \Delta N, m_0 : m_0 + \Delta M)$ and $B(m_0 : m_0 + \Delta M, l_0 : l_0 + \Delta L)$ into shared memory, sequentially for $m_0 = i\Delta M, i = 0, \dots, M/\Delta M$.

 - ▶ $\Delta N = \Delta L = \Delta M = 16 : (\Delta N + \Delta L)\Delta M \cdot 4 = 2kB$

Block Shared Memory

- ▶ shared memory is declared using the `__shared__` specifier.

```
1   __shared__ float A_tile[DN * DM];
```

- ▶ to transfer data from GPU memory to SM memory, it needs to be **cooperatively loaded** by the threads.
 - ▶ each thread is loading some part.
 - ▶ before using the shared data, it must be ensured that all threads have completed the loading steps.
 - ▶ all threads of a block have to be synchronized.
 - ▶ all threads of a block can be barrier synchronized using `__syncthreads()`.

```
1   __syncthreads();
```

- ▶ for tiled matrix multiplication, each thread $C(n, l)$ will load
 - ▶ a $\Delta M / \Delta L$ row fragment of tile row $A(n, m_0 : m_0 + \Delta M)$ and
 - ▶ a $\Delta M / \Delta N$ column fragment of tile column $B(m_0 : m_0 + \Delta M, l)$.

Example: Matrix Multiplication (GPU, tiled)

```

1  #include "Matrix.h"
2
3  const int DN = 16, DL = 16, DM = 16;

29 void mult(Matrix& A, Matrix& B, Matrix& C) {
30     const int N = A._N, M = A._M, L = B._M;
31
32     float *d_A, *d_B, *d_C;
33     cudaMalloc(&d_A, N*M*sizeof(float));
34     cudaMalloc(&d_B, M*L*sizeof(float));
35     cudaMalloc(&d_C, N*L*sizeof(float));
36     cudaMemcpy(d_A, A._data, N*M*sizeof(float), cudaMemcpyHostToDevice);
37     cudaMemcpy(d_B, B._data, M*L*sizeof(float), cudaMemcpyHostToDevice);
38
39     dim3 block(DN, DL), grid(N/DN, L/DL);
40     d_mult<<<grid, block>>>(N, M, L, d_A, d_B, d_C);
41     cudaDeviceSynchronize();
42     cudaMemcpy(C._data, d_C, N*L*sizeof(float), cudaMemcpyDeviceToHost);
43     cudaFree(d_A);
44     cudaFree(d_B);
45     cudaFree(d_C);
46 }
47
48
49 int main(int argn, char** argv) {
50     const int N = 4096, M = 2048, L = 2048;
51     Matrix A(N,M), B(M,L), C(N,L);
52     mult(A, B, C);
53 }

```

Example: Matrix Multiplication (GPU, tiled)

```

1 #include "Matrix.h"
2
3 const int DN = 16, DL = 16, DM = 16;
4
5 __global__ void d_mult(int N, int M, int L,
6     float * A, float * B, float * C) {
7     __shared__ float A_tile [DN * DM];
8     __shared__ float B_tile [DM * DL];
9     int n0 = blockIdx.x, dn = threadIdx.x,
10        l0 = blockIdx.y, dl = threadIdx.y;
11     int n = n0 * DN + dn;
12     int l = l0 * DL + dl;
13     float c = 0;
14     for (int m0 = 0; m0 < M/DM; ++m0) {
15         int DM_n = DM/DL, DM_l = DM/DN;
16         for (int dm = dn*DM_n; dm < (dn+1)*DM_n; ++dm)
17             A_tile[dn * DM + dm] = A[n*M + m0*DM + dm];
18         for (int dm = dn*DM_l; dm < (dn+1)*DM_l; ++dm)
19             B_tile[dm * DL + dl] = B[(m0*DM + dm)*L + l];
20         __syncthreads();
21
22         for (int dm = 0; dm < DM; ++dm)
23             c += A_tile[dn*DM + dm] * B_tile[dm*DL + dl];
24         __syncthreads();
25     }
26     C[n*L + l] = c;
27 }
  
```


Example: Matrix Multiplication (GPU, tiled)

not using shared memory: using shared memory:

```

1 #include "Matrix.h"
2
3 const int DN = 16, DL = 16, DM = 16;
4
5 __global__ void d_mult(int N, int M, int L,
6     float* A, float* B, float* C) {
7     int n0 = blockIdx.x, dn = threadIdx.x,
8         l0 = blockIdx.y, dl = threadIdx.y;
9     int n = n0 * DN + dn;
10    int l = l0 * DL + dl;
11    float c = 0;
12    for (int m0 = 0; m0 < M/DM; ++m0)
13        for (int dm = 0; dm < DM; ++dm)
14            c += A[n*M + m0*DM + dm]
15                * B[(m0*DM + dm)*L + l];
16    C[n*L + l] = c;
17 }

```

```

1 #include "Matrix.h"
2
3 const int DN = 16, DL = 16, DM = 16;
4
5 __global__ void d_mult(int N, int M, int L,
6     float* A, float* B, float* C) {
7     __shared__ float A_tile [DN * DM];
8     __shared__ float B_tile [DM * DL];
9     int n0 = blockIdx.x, dn = threadIdx.x,
10        l0 = blockIdx.y, dl = threadIdx.y;
11    int n = n0 * DN + dn;
12    int l = l0 * DL + dl;
13    float c = 0;
14    for (int m0 = 0; m0 < M/DM; ++m0) {
15        int DM_n = DM/DL, DM_l = DM/DN;
16        for (int dm = dn*DM_n; dm < (dn+1)*DM_n; ++dm)
17            A_tile[dn * DM + dm] = A[n*M + m0*DM + dm];
18        for (int dm = dn*DM_l; dm < (dn+1)*DM_l; ++dm)
19            B_tile[dm * DL + dl] = B[(m0*DM + dm)*L + dl];
20        __syncthreads();
21
22        for (int dm = 0; dm < DM; ++dm)
23            c += A_tile[dn*DM + dm] * B_tile[dm*DL + dl];
24        __syncthreads();
25    }
26    C[n*L + l] = c;
27 }

```

Remarks

- ▶ the example works only as long as all size ratios are integer.
 - ▶ $N/\Delta N$, $L/\Delta L$, $M/\Delta M$, $\Delta M/\Delta N$, $\Delta M/\Delta L$
 - ▶ otherwise memory accesses have to be guarded.

Summary (1/2)

- ▶ GPUs provide **massively parallel** computation for little money.
 - ▶ 3000–5000 cores per card
- ▶ GPUs support **Single Instruction Multiple Data (SIMD)** parallelism.
 - ▶ for smaller number of threads (usually 32; **warps**)
- ▶ **Compute Unified Device Architecture (CUDA)** provides
 - ▶ a preprocessor and
 - ▶ an API for GPU-enhanced programs in C, C++ and Fortran.
- ▶ The GPU program abstraction is a **kernel** that is run over a 1 to 3 dimensional cartesian product of integer ranges (**elements**, aka **threads**).
 - ▶ these elements usually denote indices into a data array.
- ▶ To exchange data between CPU and GPU,
 - ▶ either managed **unified memory** can be used or
 - ▶ data is explicitly transferred in before and after GPU computations.

Summary (2/2)

- ▶ Elements/indices/threads are grouped in **blocks** of consecutive values.
 - ▶ all threads of a block will run on the same streaming multiprocessor
- ▶ threads within a block run in **warps** of 32 threads in lockstep
 - ▶ if their control flow diverges, the warps are split accordingly and run sequentially.
 - ▶ thus their control flow should diverge as little as possible.
- ▶ threads within a block can access **shared memory**.
 - ▶ much faster access.
 - ▶ useful when input data of different threads overlaps.
 - ▶ data needs to be **cooperatively loaded** from GPU memory.

Further Readings

- ▶ There are many very good lectures and tutorials collected here:
<https://developer.nvidia.com/educators/existing-courses>

References I

Michael J Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 100(9): 948–960, 1972.