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.	9.4.	(1)	0. Introduction
			A. Parallel Computing
Tue.	16.4.	(2)	A.1 Threads
Tue.	23.4.	(3)	A.2 Message Passing Interface (MPI)
Tue.	30.4.	(4)	A.3 Graphical Processing Units (GPUs)
			B. Distributed Storage
Tue.	7.5.	(5)	B.1 Distributed File Systems
Tue.	14.5.	(6)	B.2 Partioning of Relational Databases
Tue.	21.5.	(7)	B.3 NoSQL Databases
			C. Distributed Computing Environments
Tue.	28.5.	(8)	C.1 Map-Reduce
Tue.	4.6.	—	— *Pentecoste Break* —
Tue.	11.6.	(9)	C.2 Resilient Distributed Datasets (Spark)
Tue.	18.6.	(10)	C.3 Computational Graphs (TensorFlow)
			D. Distributed Machine Learning Algorithms
Tue.	25.6.	(11)	D.1 Distributed Stochastic Gradient Descent
Tue.	2.7.	(12)	D.2 Distributed Matrix Factorization
Tue.	9.7.	(13)	Questions and Answers

Outline

1. GPUs vs CPUs

2. Basics of GPU Programming

3. Example: Color to Grayscale

4. Example: Matrix Multiplication

5. Block Shared Memory

Outline

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: ▶ old single-core CPUs	MISD: ▶ unusual
	multiple	SIMD: ▶ GPUs ▶ SIMD operations in ordinary CPUs	MIMD: ▶ 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.
Turing	2018	64	8	64–96	
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 RTX	Turing	1	72	64	4608	
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 RTX 2080 Ti	Turing	1	68	64	4352	
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

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

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:

$$\begin{aligned}\text{elem}.x &:= \text{blockDim}.x \cdot \text{blockIdx}.x + \text{threadIdx}.x\\ \text{elem}.y &:= \text{blockDim}.y \cdot \text{blockIdx}.y + \text{threadIdx}.y\\ \text{elem}.z &:= \text{blockDim}.z \cdot \text{blockIdx}.z + \text{threadIdx}.z\end{aligned}$$

  - the total grid size is

$$\begin{aligned}\text{size}.x &:= \text{blockDim}.x \cdot \text{gridDim}.x\\ \text{size}.y &:= \text{blockDim}.y \cdot \text{gridDim}.y\\ \text{size}.z &:= \text{blockDim}.z \cdot \text{gridDim}.z\end{aligned}$$

CPU/GPU Cooperation

1. allocate unified memory on GPU/CPU

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

2. start GPU computation

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

3. wait for GPU program to complete

```
cudaDeviceSynchronize();
```

4. free memory

```
cudaFree(ptr);
```

Kernel Calls

code:

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

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

```
gridDim.x = numberOfBlocks; blockDim.x = sizeOfBlocks;
for blockIdx.x = 0 ... numberOfBlocks.x−1 in parallel:
  for threadIdx.x = 0 ... sizeOfBlocks.x−1 in parallel :
    kernel(kernelParams ...);
```

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

```
gridDim = numberOfBlocks; blockDim = sizeOfBlocks;
for blockIdx.x = 0 ... numberOfBlocks.x−1 in parallel:
  for blockIdx.y = 0 ... numberOfBlocks.y−1 in parallel:
    for blockIdx.z = 0 ... numberOfBlocks.z−1 in parallel:
      for threadIdx.x = 0 ... sizeOfBlocks.x−1 in parallel :
        for threadIdx.y = 0 ... sizeOfBlocks.y−1 in parallel :
          for threadIdx.z = 0 ... sizeOfBlocks.z−1 in parallel :
            kernel(kernelParams ...);
```

Example: Add Two Vectors

```
#include <iostream>
#include <math.h>
#define CEIL(x,y) (x+y-1)/y

__global__ void add(float *x, float *y) {
  int n = blockIdx.x * blockDim.x + threadIdx.x;
  y[n] = x[n] + y[n];
}

int main(void) {
  int N = 1<<20;
  float *x, *y;
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));
  for (int n = 0; n < N; n++) {
    x[n] = 1.0f;
    y[n] = 2.0f;
  }

  add<<<CEIL(N, 256), 256>>>(x, y);
  cudaDeviceSynchronize();

  float maxError = 0.0f;
  for (int n = 0; n < N; n++)
    maxError = fmax(maxError, fabs(y[n]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  cudaFree(x);
  cudaFree(y);
  return 0;
}
```

Compiling CUDA Code

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

```
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 rangesize/blocksize 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)

```
#include <iostream>
#include <math.h>
#define CEIL(x,y) (x+y-1)/y

__global__ void add(float *x, float *y, int N) {
  int n = blockIdx.x * blockDim.x + threadIdx.x;
  if (n < N)
    y[n] = x[n] + y[n];
}

int main(void) {
  int N = 1<<20;
  float *x, *y;
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));
  for (int n = 0; n < N; n++) {
    x[n] = 1.0f;
    y[n] = 2.0f;
  }

  add<<<CEIL(N, 198), 198>>>(x, y, N);
  cudaDeviceSynchronize();

  float maxError = 0.0f;
  for (int n = 0; n < N; n++)
    maxError = fmax(maxError, fabs(y[n]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  cudaFree(x);
  cudaFree(y);
  return 0;
```

Outline

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

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

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

```
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

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

4b. free GPU memory

```
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

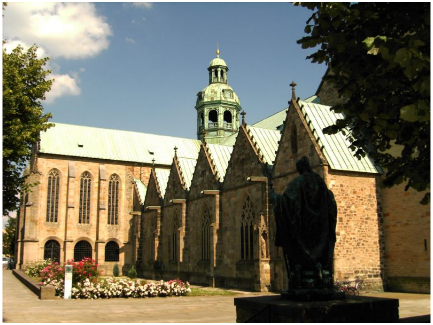
$$\text{index}(r, c, h) := (r \cdot C + c) \cdot H + h$$

Example: Color to Grayscale

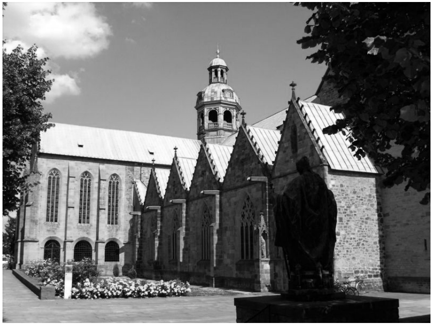
```
#include <opencv2/highgui/highgui.hpp>

#define DTYPE unsigned char

__global__ void rgb2gray(const DTYPE* in, DTYPE* out) {
  int index_out = blockIdx.x * blockDim.x + threadIdx.x;
  int index_in = index_out * 3;
  out[index_out] = (in[index_in] + in[index_in + 1] + in[index_in + 2]) / 3;
}

int main(int arg, char* args[]) {
  cv::Mat img = cv::imread("dom_hildesheim.png", CV_LOAD_IMAGE_COLOR);
  cv::Mat img_gray (img.rows, img.cols, CV_8UC1);

  int img_size = img.rows * img.cols * img.channels() * sizeof(DTYPE),
      img_gray_size = img_gray.rows * img_gray.cols * img_gray.channels() * sizeof(DTYPE);
  DTYPE *in = 0, *out = 0;
  cudaMalloc(&in, img_size);
  cudaMalloc(&out, img_gray_size);
  cudaMemcpy(in, img.data, img_size, cudaMemcpyHostToDevice);

  rgb2gray<<<img.rows, img.cols>>>(in, out);
  cudaDeviceSynchronize();

  cudaMemcpy(img_gray.data, out, img_gray_size, cudaMemcpyDeviceToHost);
  cudaFree(in);
  cudaFree(out);
  cv::imwrite("out-gray.png", img_gray);
}
```

Example: Color to Grayscale

input image:



output image:



Example: Color to Grayscale (alternative grid)

```
#include <opencv2/highgui/highgui.hpp>

#define DTYPE unsigned char
#define CEIL(A,B) (A+B−1)/B

__global__ void rgb2gray(const DTYPE* in, DTYPE* out, int num_rows, int num_cols) {
  int row = blockIdx.x * blockDim.x + threadIdx.x,
      col = blockIdx.y * blockDim.y + threadIdx.y;
  if (row < num_rows && col < num_cols) {
    int index_out = row * num_cols + col;
    int index_in = index_out * 3;
    out[index_out] = (in[index_in] + in[index_in + 1] + in[index_in + 2]) / 3;
  }
}

int main(int arg, char* args []) {
  cv::Mat img = cv::imread("dom_hildesheim.png", CV_LOAD_IMAGE_COLOR);
  cv::Mat img_gray (img.rows, img.cols, CV_8UC1);

  int img_size = img.rows * img.cols * img.channels() * sizeof (DTYPE),
      img_gray_size = img_gray.rows * img_gray.cols * img_gray.channels() * sizeof(DTYPE);
  DTYPE *in = 0, *out = 0;
  cudaMalloc(&in, img_size);
  cudaMalloc(&out, img_gray_size);
  cudaMemcpy(in, img.data, img_size, cudaMemcpyHostToDevice);

  rgb2gray<<<dim3(CEIL(img.rows,32), CEIL(img.cols,32)), dim3(32,32)>>>(in, out, img.rows, img.cols);
  cudaDeviceSynchronize ();

  cudaMemcpy(img_gray.data, out, img_gray_size, cudaMemcpyDeviceToHost);
  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,...,Turing; 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 priorization
- 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

Example: Matrix Multiplication (CPU, direct)

```
#include <stdlib.h>

class Matrix {
public:
  int _N, _M;
  float* _data;

  Matrix(int N, int M)
    : _N(N), _M(M), _data(new float[N*M]) {
    for (int n = 0; n < N; ++n)
      for (int m = 0; m < M; ++m)
        _data[n*_M + m] = rand() * 2.0f
                                   / RAND_MAX - 1.0f;
  }

  float& operator()(int n, int m) {
    return _data[n*_M + m];
  }
};
```

```
#include "Matrix.h"

void mult(Matrix& A, Matrix& B, Matrix& C) {
  const int N = A._N, M = A._M, L = B._M;
  for (int n = 0; n < N; ++n) {
    for (int l = 0; l < L; ++l) {
      float c = 0;
      for (int m = 0; m < M; ++m)
        c += A(n,m) * B(m,l);
      C(n,l) = c;
    }
  }
}

int main(int argn, char** argv) {
  const int N = 4096, M = 2048, L = 2048;
  Matrix A(N,M), B(M,L), C(N,L);
  mult(A, B, C);
}
```

Example: Matrix Multiplication (CPU, tiled)

```
#include "Matrix.h"
#include <cmath>
#include <algorithm>

void mult(Matrix& A, Matrix& B, Matrix& C) {
  const int N = A._N, M = A._M, L = B._M, K = ceil(sqrt(M));
  for (int n = 0; n < N; ++n)
    for (int l = 0; l < L; ++l)
      C(n,l) = 0;
  for (int n0 = 0; n0 < N; n0+= K) {
    for (int l0 = 0; l0 < L; l0+= K) {
      for (int m0 = 0; m0 < M; m0+= K) {
        for (int n = n0; n < std::min(N, n0+K); ++n) {
          for (int l = l0; l < std::min(L, l0+K); ++l) {
            float c = 0;
            for (int m = m0; m < std::min(M, m0+K); ++m)
              c += A(n,m) * B(m,l);
            C(n,l) += c;
          }
        }
      }
    }
  }
}

int main(int argn, char** argv) {
  const int N = 4096, M = 2048, L = 2048;
  Matrix A(N,M), B(M,L), C(N,L);
  mult(A, B, C);
}
```

Example: Matrix Multiplication (GPU, direct)

```
#include "Matrix.h"

__global__ void d_mult(int M, int L, float* A, float* B, float* C) {
  int n = blockIdx.x * blockDim.x + threadIdx.x;
  int l = blockIdx.y * blockDim.y + threadIdx.y;
  float c = 0;
  for (int m = 0; m < M; ++m)
    c += A[n*M + m] * B[m*L + l];
  C[n*L + l] = c;
}

void mult(Matrix& A, Matrix& B, Matrix& C) {
  const int N = A._N, M = A._M, L = B._M;

  float *d_A, *d_B, *d_C;
  cudaMalloc(&d_A, N*M*sizeof(float));
  cudaMalloc(&d_B, M*L*sizeof(float));
  cudaMalloc(&d_C, N*L*sizeof(float));
  cudaMemcpy(d_A, A._data, N*M*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B._data, M*L*sizeof(float), cudaMemcpyHostToDevice);

  dim3 block(16, 16), grid(N/16, L/16);
  d_mult<<<grid, block>>>(M, L, d_A, d_B, d_C);
  cudaDeviceSynchronize();
  cudaMemcpy(C._data, d_C, N*L*sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}
```

```
int main(int argn, char** argv) {
  const int N = 4096, M = 2048, L = 2048;
  Matrix A(N,M), B(M,L), C(N,L);
  mult(A, B, C);
}
```

Outline

Example: Matrix Multiplication (GPU, tiled, v0)

```
#include "Matrix.h"

const int DN = 16, DL = 16, DM = 16;

__global__ void d_mult(int N, int M, int L,
                    float* A, float* B, float* C) {
  int n0 = blockIdx.x, dn = threadIdx.x,
      l0 = blockIdx.y, dl = threadIdx.y;
  int n = n0 * DN + dn;
  int l = l0 * DL + dl;
  float c = 0;
  for (int m0 = 0; m0 < M/DM; ++m0)
    for (int dm = 0; dm < DM; ++dm)
      c += A[n*M + m0*DM + dm]
          * B[(m0*DM + dm)*L + l];
  C[n*L + l] = c;
}

void mult(Matrix& A, Matrix& B, Matrix& C) {
  const int N = A._N, M = A._M, L = B._M;

  float *d_A, *d_B, *d_C;
  cudaMalloc(&d_A, N*M*sizeof(float));
  cudaMalloc(&d_B, M*L*sizeof(float));
  cudaMalloc(&d_C, N*L*sizeof(float));
  cudaMemcpy(d_A, A._data, N*M*sizeof(float), cudaMe
  cudaMemcpy(d_B, B._data, M*L*sizeof(float), cudaMe

  dim3 block(DN, DL), grid(N/DN, L/DL);
  d_mult<<<grid, block>>>(N, M, L, d_A, d_B, d_C
  cudaDeviceSynchronize();
  cudaMemcpy(C._data, d_C, N*L*sizeof(float), cudaMe
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}


int main(int argn, char** argv) {
  const int N = 4096, M = 2048, L = 2048;
  Matrix A(N,M), B(M,L), C(N,L);
  mult(A, B, C);
}
```

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 $m = 0, \ldots, M - 1$

- all threads $C(n, l)$ and $C(n, l')$ share M of those
  - $A(n, m)$ for $m = 0, \ldots, M - 1$
- all threads $C(n, l)$ and $C(n', l)$ share M of those
  - $B(m, l)$ for $m = 0, \ldots, M - 1$

- 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 $m = 0, \ldots, M-1$
- all threads $C(n,l)$ and $C(n,l')$ share M of those
  - $A(n,m)$ for $m = 0, \ldots, M-1$
- all threads $C(n,l)$ and $C(n',l)$ share M of those
  - $B(m,l)$ for $m = 0, \ldots, M-1$
- 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 $m_0 = i\Delta M, i = 0, \ldots, M/\Delta M$.
  - $\Delta N = \Delta L = \Delta M = 16 : (\Delta N + \Delta L)\Delta M \cdot 4 = 2kB$

Block Shared Memory

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

```
        __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()`.

```
        __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)

```
#include "Matrix.h"

const int DN = 16, DL = 16, DM = 16;


void mult(Matrix& A, Matrix& B, Matrix& C) {
  const int N = A._N, M = A._M, L = B._M;

  float *d_A, *d_B, *d_C;
  cudaMalloc(&d_A, N*M*sizeof(float));
  cudaMalloc(&d_B, M*L*sizeof(float));
  cudaMalloc(&d_C, N*L*sizeof(float));
  cudaMemcpy(d_A, A._data, N*M*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B._data, M*L*sizeof(float), cudaMemcpyHostToDevice);

  dim3 block(DN, DL), grid(N/DN, L/DL);
  d_mult<<<grid, block>>>(N, M, L, d_A, d_B, d_C);
  cudaDeviceSynchronize();
  cudaMemcpy(C._data, d_C, N*L*sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}


int main(int argn, char** argv) {
  const int N = 4096, M = 2048, L = 2048;
  Matrix A(N,M), B(M,L), C(N,L);
  mult(A, B, C);
}
```

Example: Matrix Multiplication (GPU, tiled)

```
#include "Matrix.h"

const int DN = 16, DL = 16, DM = 16;

__global__ void d_mult(int N, int M, int L,
                       float* A, float* B, float* C) {
  __shared__ float A_tile [DN * DM];
  __shared__ float B_tile [DM * DL];
  int n0 = blockIdx.x, dn = threadIdx.x,
      l0 = blockIdx.y, dl = threadIdx.y;
  int n = n0 * DN + dn;
  int l = l0 * DL + dl;
  float c = 0;
  for (int m0 = 0; m0 < M/DM; ++m0) {
    int DM_n = DM/DL, DM_l = DM/DN;
    for (int dm = dn*DM_n; dm < (dn+1)*DM_n; ++dm)
      A_tile[dn * DM + dm] = A[n*M + m0*DM + dm];
    for (int dm = dn*DM_l; dm < (dn+1)*DM_l; ++dm)
      B_tile[dm * DL + dl] = B[(m0*DM + dm)*L + l];
    __syncthreads();

    for (int dm = 0; dm < DM; ++dm)
      c += A_tile[dn*DM + dm] * B_tile[dm*DL + dl];
    __syncthreads();
  }
  C[n*L + l] = c;
}
```

Example: Matrix Multiplication (GPU, tiled)

not using shared memory:

```
#include "Matrix.h"

const int DN = 16, DL = 16, DM = 16;

__global__ void d_mult(int N, int M, int L,
                       float* A, float* B, float* C) {
  int n0 = blockIdx.x, dn = threadIdx.x,
      l0 = blockIdx.y, dl = threadIdx.y;
  int n = n0 * DN + dn;
  int l = l0 * DL + dl;
  float c = 0;
  for (int m0 = 0; m0 < M/DM; ++m0)
    for (int dm = 0; dm < DM; ++dm)
      c += A[n*M + m0*DM + dm]
           * B[(m0*DM + dm)*L + l];
  C[n*L + l] = c;
}
```

using shared memory:

```
#include "Matrix.h"

const int DN = 16, DL = 16, DM = 16;

__global__ void d_mult(int N, int M, int L,
                       float* A, float* B, float* C) {
  __shared__ float A_tile [DN * DM];
  __shared__ float B_tile [DM * DL];
  int n0 = blockIdx.x, dn = threadIdx.x,
      l0 = blockIdx.y, dl = threadIdx.y;
  int n = n0 * DN + dn;
  int l = l0 * DL + dl;
  float c = 0;
  for (int m0 = 0; m0 < M/DM; ++m0) {
    int DM_n = DM/DL, DM_l = DM/DN;
    for (int dm = dn*DM_n; dm < (dn+1)*DM_n; +
      A_tile[dn * DM + dm] = A[n*M + m0*DM + d
    for (int dm = dn*DM_l; dm < (dn+1)*DM_l; +
      B_tile[dm * DL + dl] = B[(m0*DM + dm)*L +
    __syncthreads();

    for (int dm = 0; dm < DM; ++dm)
      c += A_tile[dn*DM + dm] * B_tile[dm*DL + c
    __syncthreads();
  }
  C[n*L + l] = c;
}
```

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 litte 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 transfered 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.