Parallel Computing

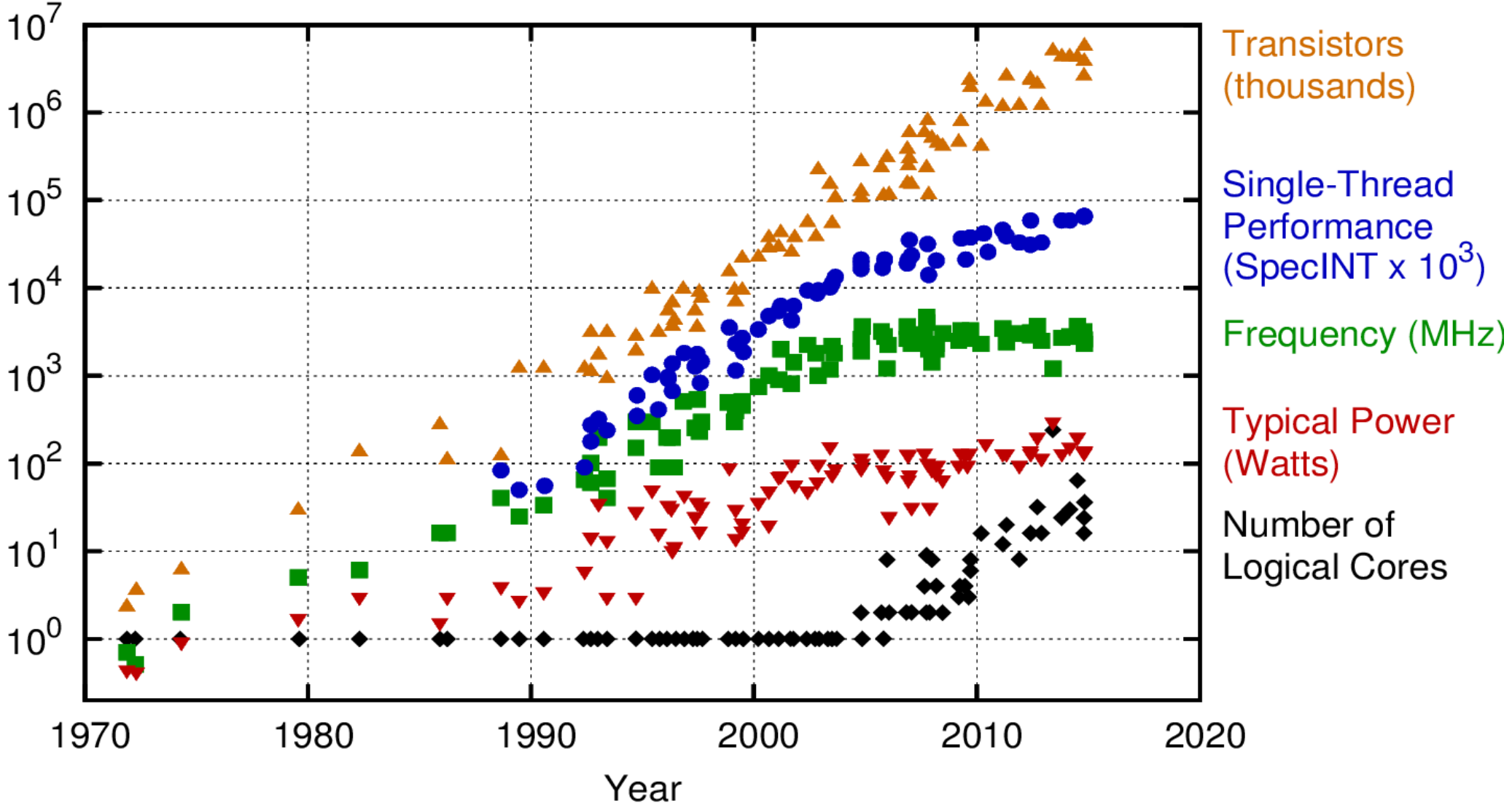
Dominik Adamski

Łódź, 21.02.2018

Content

- Motivation
- OpenMP
- CUDA
- Tiling optimization
- General guidelines
- Literature

Motivation



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Motivating Example

```
1  using namespace std;
2  typedef array<array<float, 256>, 256>  Matrix;
3
4  unique_ptr<Matrix> mm_seq(const unique_ptr<Matrix> & a,
5                            const unique_ptr<Matrix> & b) {
6      unique_ptr <Matrix> c (new Matrix);
7      for (int i = 0; i < 256; ++i) {
8          for (int j = 0; j < 256; ++j) {
9              float element = 0;
10             for (int k = 0; k < 256; ++k) {
11                 element += (*a)[i][k] * (*b)[k][j];
12             }
13             (*c)[i][j] = element;
14         }
15     }
16     return c;
17 }
```

- Easy for parallelization
- All loops can be parallelized
- One of the basic mathematical operation

OpenMP

- Set of compiler pragmas, library routines and environment variable
- Support for multi-platform shared memory multiprocessing programming in C/C++ and Fortran
- Cross platform
- Current stable version 4.5 (added support for offloading)
- Available for gcc, clang, icc, msvc (only version 2.0)
- Support parallelization on multiple levels (simd, loops, tasks)

OpenMP vs C++ threads

OpenMP

- Minimal code refactoring
- Limited application range
- Required compiler support

C++ threads

- Large code refactoring
- Broad application range
- Part of C++11 standard

OpenMP – Example 1/2

```
4   #pragma omp parallel for
5   »   for (int i = 0; i < 256; ++i) {
6   »   »   for (int j = 0; j < 256; ++j) {
7   »   »   »   float element = 0;
8   »   »   »   for (int k = 0; k < 256; ++k) {
9   »   »   »   »   element += (*a)[i][k] * (*b)[k][j];
10  »   »   »   }
11  »   »   »   (*c)[i][j] = element;
12  »   »   }
13  »   }
14
15  //gcc -fopenmp test.c -lgomp -> parallel execution
16  //gcc test.c -> sequential execution
17
```

- Only one pragma
- Loop in line 5 parallelized
- All variables declared outside pragma are shared
- Variables declared inside pragma scope and loop counter of parallelized loop are private

OpenMP – Example 2/2

```
4     for (int i = 0; i < 256; ++i) {
5         for (int j = 0; j < 256; ++j) {
6             float element = 0;
7 #pragma omp parallel for reduction(+ : element)
8             for (int k = 0; k < 256; ++k) {
9                 element += (*a)[i][k] * (*b)[k][j];
10            }
11            (*c)[i][j] = element;
12        }
13    }
```

- Reduction → result of accumulation multiple values into one variable
- Cannot be parallelized by simple `parallel for` clause

CUDA - Introduction

- Parallel computing on GPU
- Designed for C/C++ and Fortran
- Created by Nvidia
- Available only on Nvidia GPUs
- Heterogenous programming

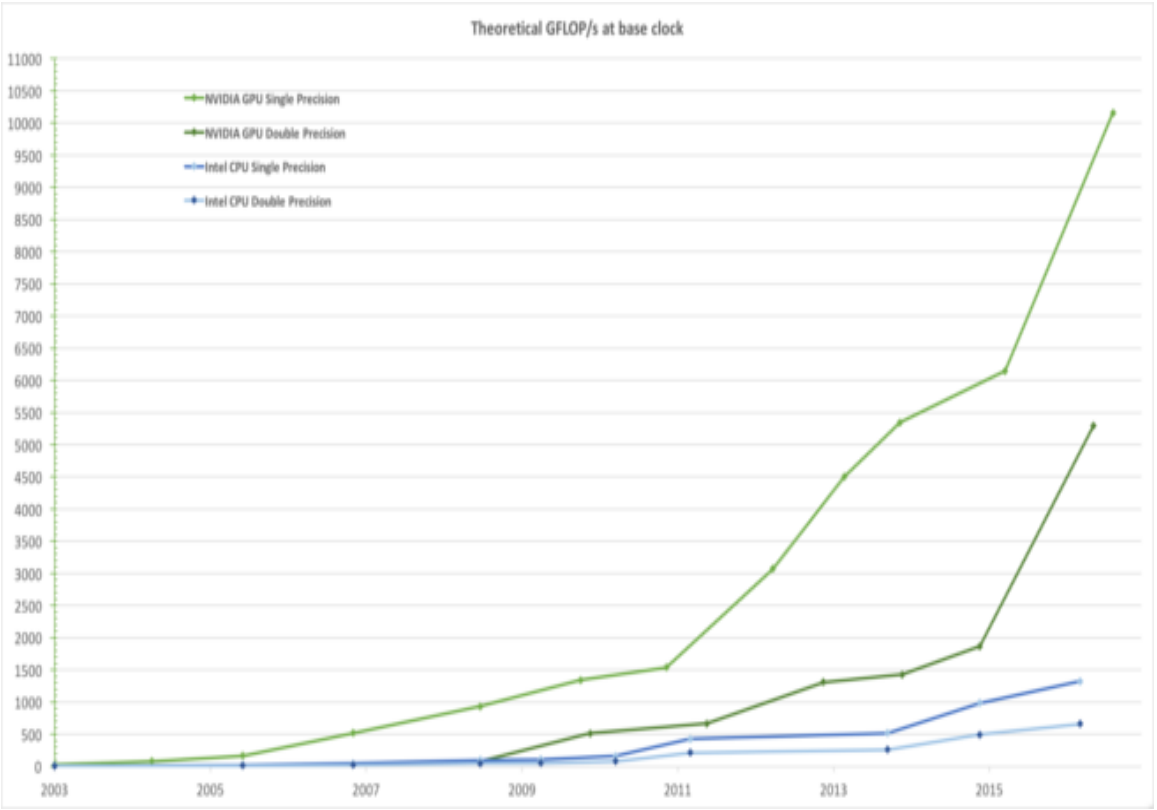
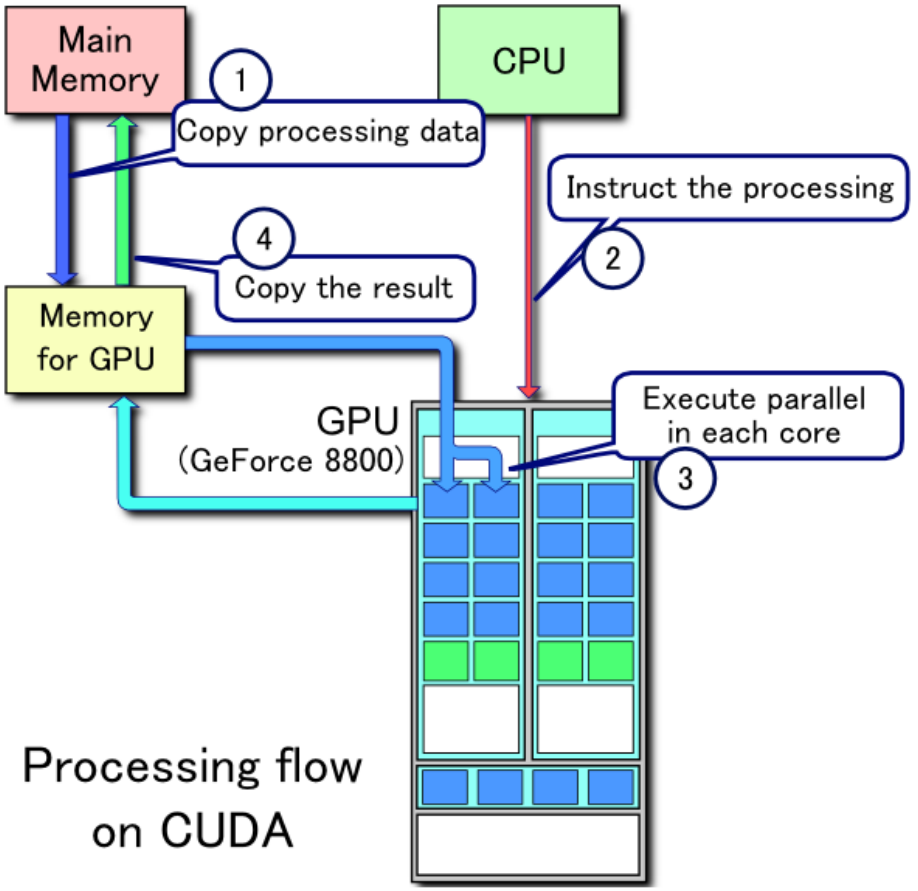


Image source: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction

CUDA – Heterogenous Programming

- Host CPU executes C program
- CUDA threads operate on physically separate GPU device
- Memory transform is needed
- Before Pascal architecture and CUDA 6 memory transform calls need to be explicitly defined



Source: https://en.wikipedia.org/wiki/CUDA

CUDA – Thread Hierarchy

- Threads are organized in blocks
- Each thread is given unique `threadIdx`
- In one block can be up to 1024 threads – hardware dependent
- Number of blocks is dependent on data size or number of processors

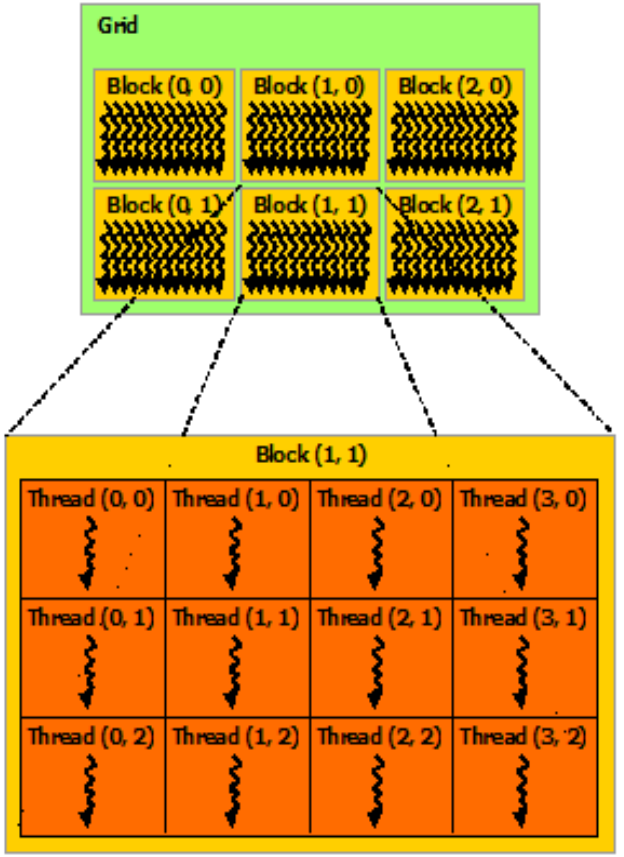


Image source: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction

CUDA – Memory Hierarchy

- Per – thread memory visible only for given thread (registers)
- Thread can access per-block and global memory
- Block memory visible only for threads in given block (L1 cache)

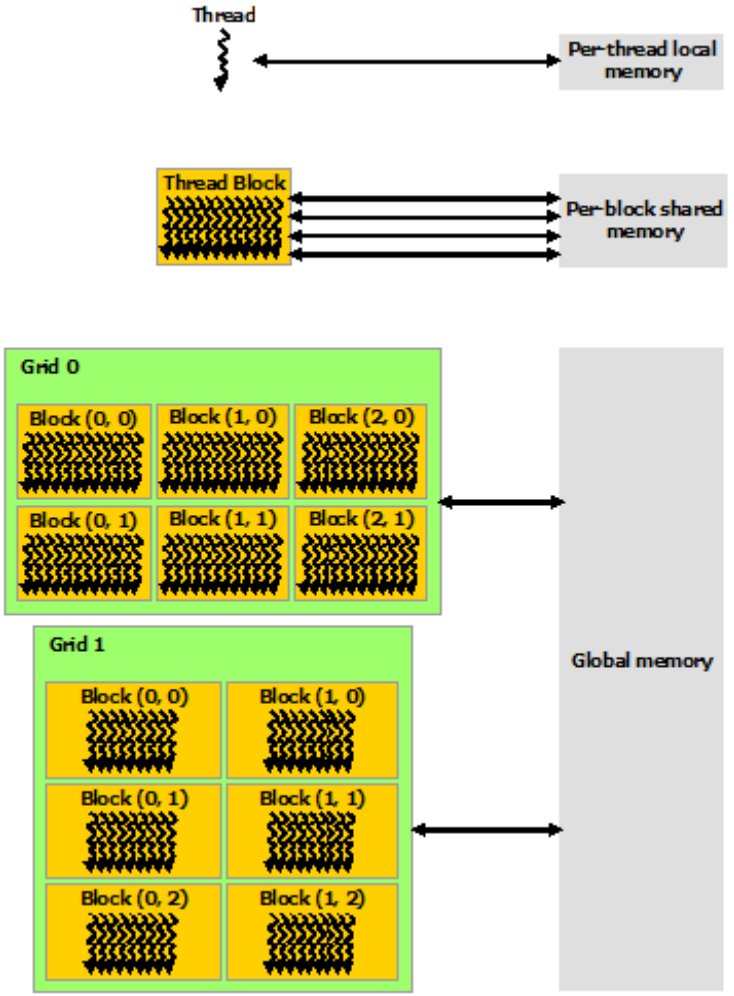


Image source:
http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction

CUDA – Host code 1/2

```
2  void cuda_multiplication (const float *mat_a,
3      float void *mat_b, float* mat_c)
4  {
5      size_t pitch_a =0, pitch_b =0, pitch_c = 0;
6      float *dev_mat_a, *dev_mat_b, *dev_mat_c;
7
8      cudaMallocPitch(&dev_mat_a, &pitch_a,
9          MATRIX_SIZE*sizeof(float), //width in bytes
10         MATRIX_SIZE); //number of rows
11     //the same for dev_mat_b, and dev_mat_c
12
13     cudaMemcpy2D(dev_mat_a, pitch_a, mat_a,
14         MATRIX_SIZE*sizeof(float), //pitch of host matrix
15         MATRIX_SIZE*sizeof(float), //width of host matrix
16         MATRIX_SIZE, cudaMemcpyHostToDevice); //ncols, type
17     //the same for dev mat b
```

- cudaMallocPitch (line 8): allocate memory on device. Recommended for 2D arrays.
- CudaMemcpy2D (line 13): transfer input data from host to device

CUDA – Host code 2/2

```
19      dim3 dimBlock(TILE_SIZE,TILE_SIZE);
20      dim3 dimGrid(MATRIX_SIZE/TILE_SIZE, MATRIX_SIZE/TILE_SIZE);
21
22      MatMulKernel<<<dimGrid, dimBlock>>>(dev_mat_a, dev_mat_b,
23              dev_mat_c,pitch_a, pitch_b, pitch_c);
24
25      cudaMemcpy2D(mat_c, MATRIX_SIZE * sizeof(float),
26              dev_mat_c, pitch_c, MATRIX_SIZE * sizeof(float),
27              MATRIX_SIZE, cudaMemcpyDeviceToHost);
28
29      cudaFree(dev_mat_a);
30      cudaFree(dev_mat_b);
31      cudaFree(dev_mat_c);
32  }
```

- Lines 19-20: definition of block and grid layout
- Lines 22-23: kernel invocation
- Lines 25-27: copy matrix result to host
- Lines 29-31: free device memory

CUDA – First kernel 1/2

```
2  __global__ void MatMulKernel(float* A, float *B,
3      float *C,size_t pitch_a, size_t pitch_b,
4      size_t  pitch_c)
5  {
6      int col = blockIdx.x * blockDim.x + threadIdx.x;
7      int row = blockIdx.y * blockDim.y + threadIdx.y;
8
9      float* c_ptr = (float*)((char*)C + row * pitch_c) + col;
10     float res = 0;
11     for (int k = 0; k < MATRIX_SIZE; ++k) {
12         float* a_ptr = (float*)((char*)A + row * pitch_a) + k;
13         float* b_ptr = (float*)((char*)B + k * pitch_b) + col;
14         res += (*a_ptr) * (*b_ptr);
15     }
16     *c_ptr = res;
17 }
```

- `__global__` (line 2) indicates function called by host
- `blockIdx` (line 6-7) block identifier
- `blockDim` (line 6-7) denotes size of block
- `threadIdx` (line 6-7) thread identifier

CUDA – First kernel 2/2

```
2  __global__ void MatMulKernel(float* A, float *B,
3      float *C,size_t pitch_a, size_t pitch_b,
4      size_t  pitch_c)
5  {
6      int col = blockIdx.x * blockDim.x + threadIdx.x;
7      int row = blockIdx.y * blockDim.y + threadIdx.y;
8
9      float* c_ptr = (float*)((char*)C + row * pitch_c) + col;
10     float res = 0;
11     for (int k = 0; k < MATRIX_SIZE; ++k) {
12         float* a_ptr = (float*)((char*)A + row * pitch_a) + k;
13         float* b_ptr = (float*)((char*)B + k * pitch_b) + col;
14         res += (*a_ptr) * (*b_ptr);
15     }
16     *c_ptr = res;
17 }
```

- `c_ptr, a_ptr, b_ptr` (lines 9, 12-13) denote pointers to global device memory
- Each thread calculates one element of output array

Tiling optimization

- Minimization of memory bottleneck
- Efficient usage of cache memory
- Beneficial for CPU and GPU computations

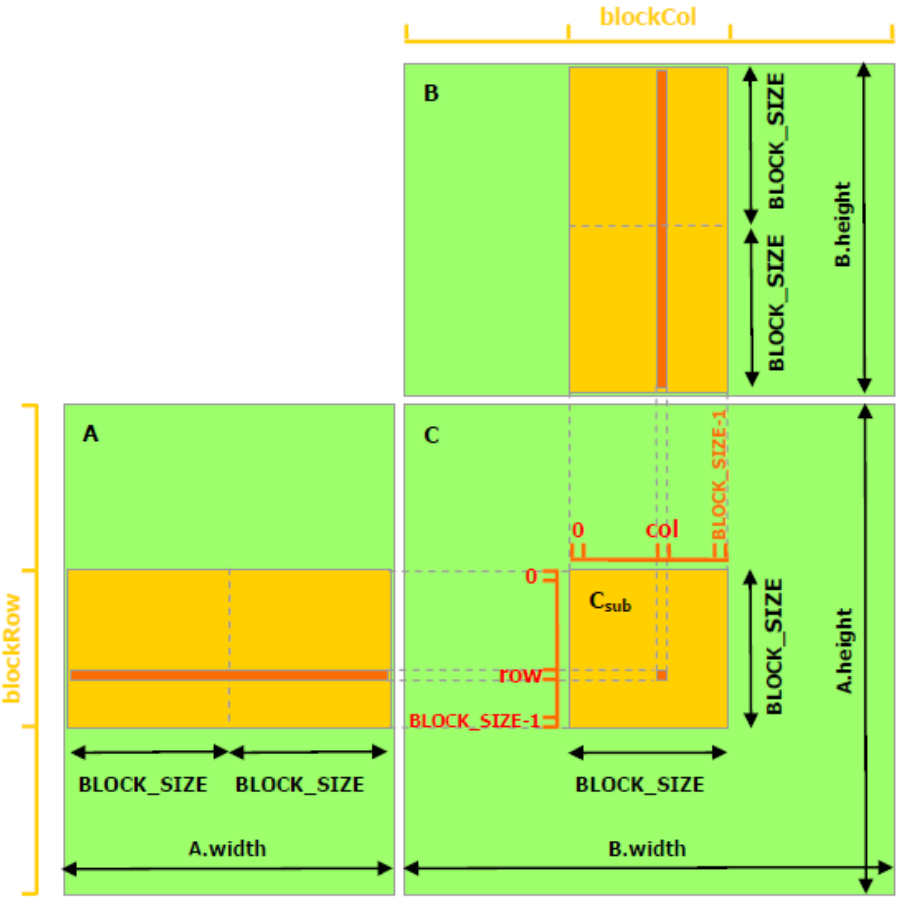


Image source: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Tiling optimization - CPU

```
1  unique_ptr<Matrix> multiply_omp_tile(const unique_ptr<Matrix> & a,
2  » » » »    » »  const unique_ptr<Matrix> & b)
3  {
4  » unique_ptr <Matrix> c (new Matrix);
5  //Assumption: MATRIX_SIZE == N * TILE_SIZE, where N is an integer
6  » for (int i = 0; i < MATRIX_SIZE; i+=TILE_SIZE)
7  » » for (int ii = i; ii < i + TILE_SIZE; ++ii)
8  » » » for (int j = 0; j < MATRIX_SIZE; j+=TILE_SIZE)
9  » » » » for (int jj = j; jj < j + TILE_SIZE; ++jj)
10 » » » » {
11 » » » » » (*c)[ii][jj] = 0;
12 » » » » » for (int k = 0; k < MATRIX_SIZE; ++k)
13 » » » » » » (*c)[ii][jj] += (*a)[ii][k] * (*b)[k][jj];
14 » » » » }
15
16 » return c;
17 }
```

- Added two additional loops (lines 7 and 9)
- Firstly calculated values inside one tile
- Size of tile dependent on hardware and software
- The best tile – minimal number of cache misses

Tiling Optimization – GPU 1/2

```
1  __device__ MATRIX_TYPE GetElement(MATRIX_TYPE* A,int row, int col,
2      size_t pitch) {
3    return *(MATRIX_TYPE*)(((char*)A + row * pitch) + col);
4  }
5
6  __global__ void MatMulKernelTiled(MATRIX_TYPE *A, MATRIX_TYPE
7      *B, MATRIX_TYPE *C, size_t pitch_a, size_t pitch_b, size_t
8      pitch_c) {
9    __shared__ MATRIX_TYPE tile_a[BLOCK_SIZE][BLOCK_SIZE];
10   __shared__ MATRIX_TYPE tile_b[BLOCK_SIZE][BLOCK_SIZE];
11
12   int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
13   int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;
14   int tmp = 0;
```

- __device__ - (line 1) function is called only inside GPU
- __shared__ (lines 9-10) tile arrays are located in fast block memory and are common for all threads inside one block

Tiling Optimization – GPU 2/2

```
16  for (int k = 0; k < gridDim.y; ++k) {
17      tile_a[threadIdx.y][threadIdx.x] = GetElement(A, row,
18          k*BLOCK_SIZE + threadIdx.x, pitch_a);
19      tile_b[threadIdx.y][threadIdx.x] = GetElement(B, k* BLOCK_SIZE
20          + threadIdx.y, col, pitch_b);
21    //Wait for finish of tile copy
22      __syncthreads();
23      for (int kk = 0; kk < BLOCK_SIZE; ++kk) {
24        tmp += tile_a[threadIdx.y][kk] * tile_b[kk][threadIdx.x];
25      }
26      //Wait until all computation is done before loading new tile
27      __syncthreads();
28    }
29  MATRIX_TYPE* c_ptr = (MATRIX_TYPE*)((char*)C + row * pitch_c) + col;
30  *c_ptr = tmp;
31  }
```

- Lines 17-20: fill tile matrix
- `__syncthreads` (lines 22, 27) – lightweight synchronization of all threads inside one block
- Lines 23-25: calculate value of one output element

General guidelines

- Profile your code to find bottlenecks
- Try to fully reuse available components
- Reduce cache misses for CPU
- Reuse fast per-block GPU memory efficiently
- Minimize number of memory transport for heterogenous computing
- Use advanced compiler tools which support aggressive code optimization (for example Polly)
- Use dedicated libraries

Further readings

- OpenMP specification: http://www.openmp.org/specifications/
- CUDA Programming Guide: http://docs.nvidia.com/cuda/cuda-c-programming-guide/
- Scott Meyer's presentation about CPU cache: https://www.youtube.com/watch?v=WDIkqP4JbkE
- Introduction to Data-Oriented Design: http://www.dice.se/wp-content/uploads/2014/12/Introduction_to_Data-Oriented_Design.pdf

Thank you for your attention.