Matrix Multiplication with OpenCL

22.11.2017 | Wolfram Schenck | Faculty of Engineering and Mathematics,
Bielefeld University of Applied Sciences,
Bielefeld, Germany

Outline

- **Learning goals:**
  - Gain insights how memory access patterns and the usage of local memory influence performance on different types of devices
  - Learn to apply explicit vectorization and other performance tuning techniques within the OpenCL framework

- **Simple matrix multiplication**
  - Memory layout
  - Explicit vectorization
- **Matrix multiplication with local memory**
  - Performance tuning
  - Performance evaluation

Matrix Multiplication

Matrix Multiplication Basics

- Matrix mult.: **C = A * B**
- Matrix sizes:
  Size of A: $m_A * n_A$
  (# of rows * # of columns)
  (height * width)
  Size of B: $m_B * n_B$
  Size of C: $m_C * n_C$
- Precondition: $n_A = m_B$
- In result: $m_C = m_A$,
  $n_C = n_B$
- Formula:

$$c_{i,j} = \sum_{e=0}^{n_A-1} a_{i,e} b_{e,j}$$

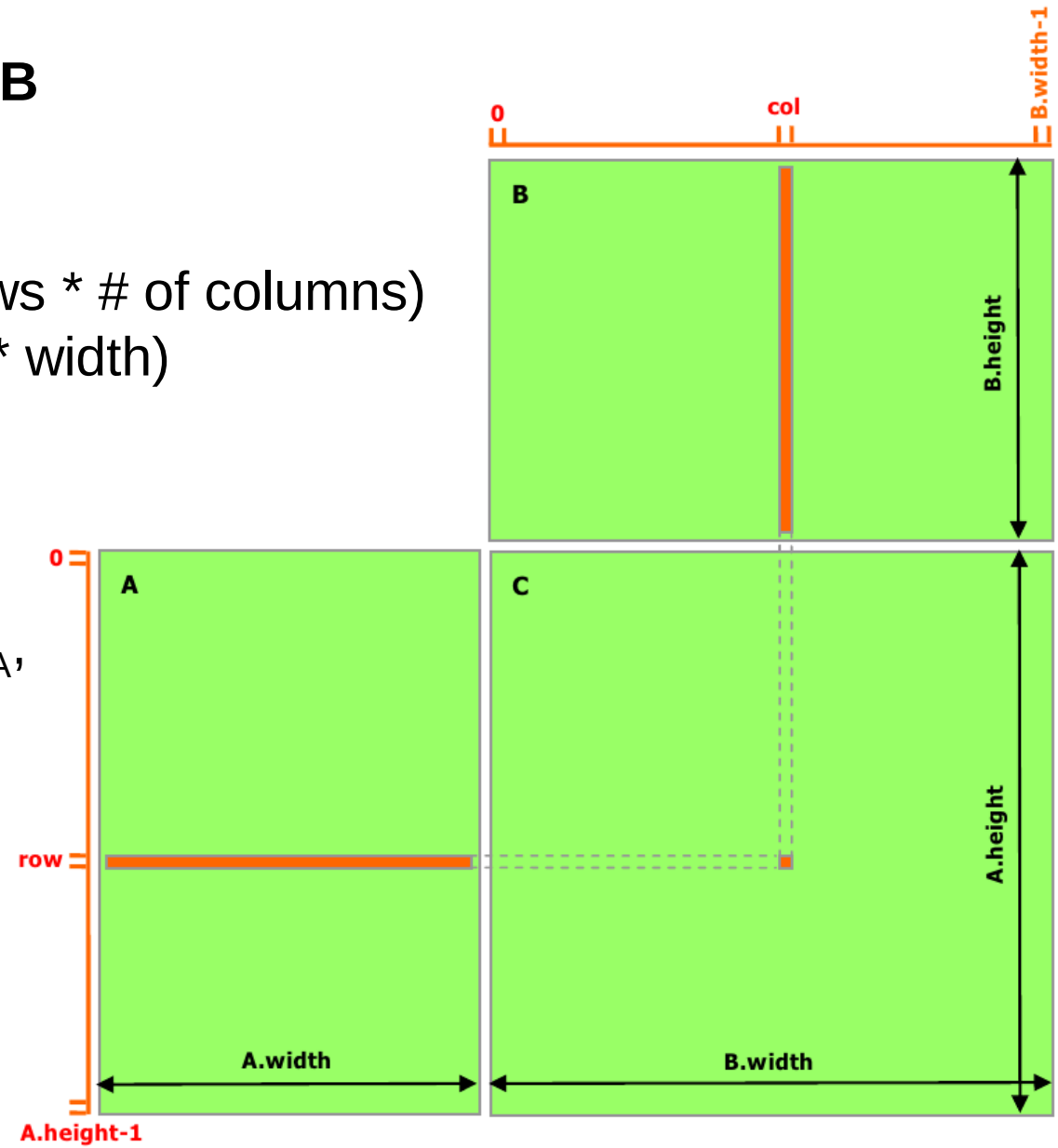
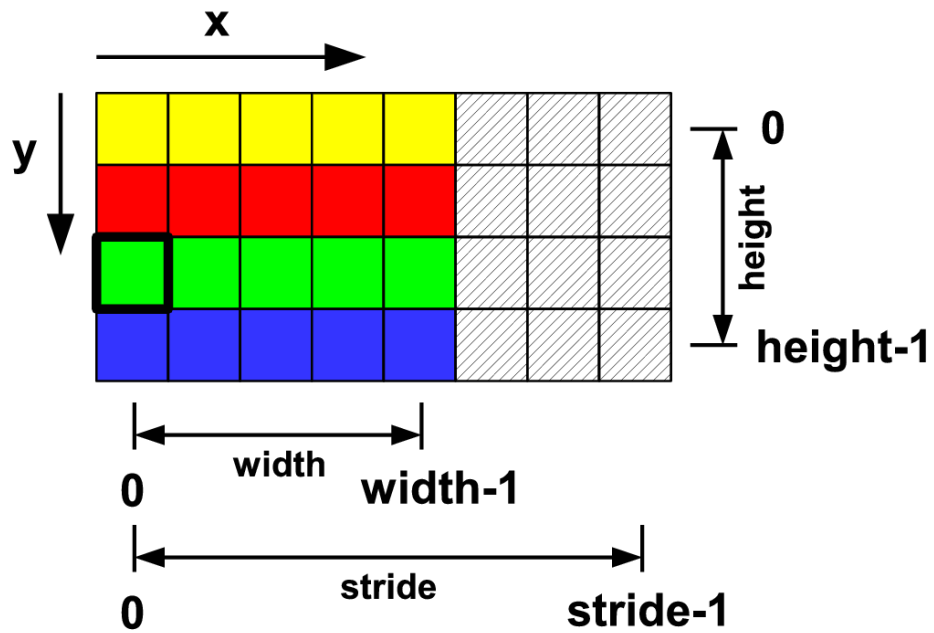


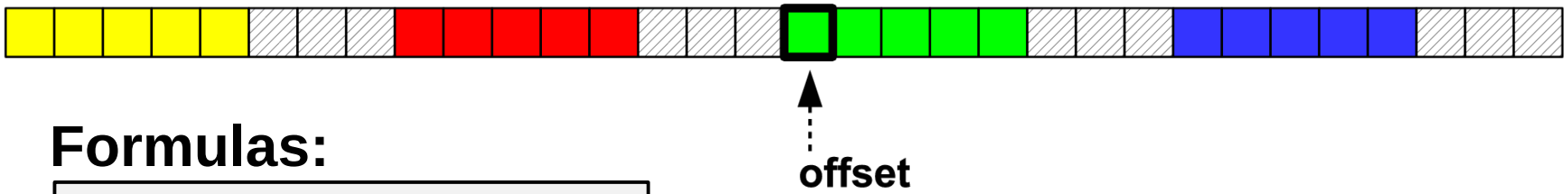
Fig.: NVIDIA

2D Arrays: Adress/Index Computation

(row-major, with stride)



IN MEMORY:



Formulas:

```
offset = y * stride + x

x = offset % stride
y = offset / stride
```

(using integer arithmetics, indices start with 0)

Simple Matrix Multiplication: Host Code

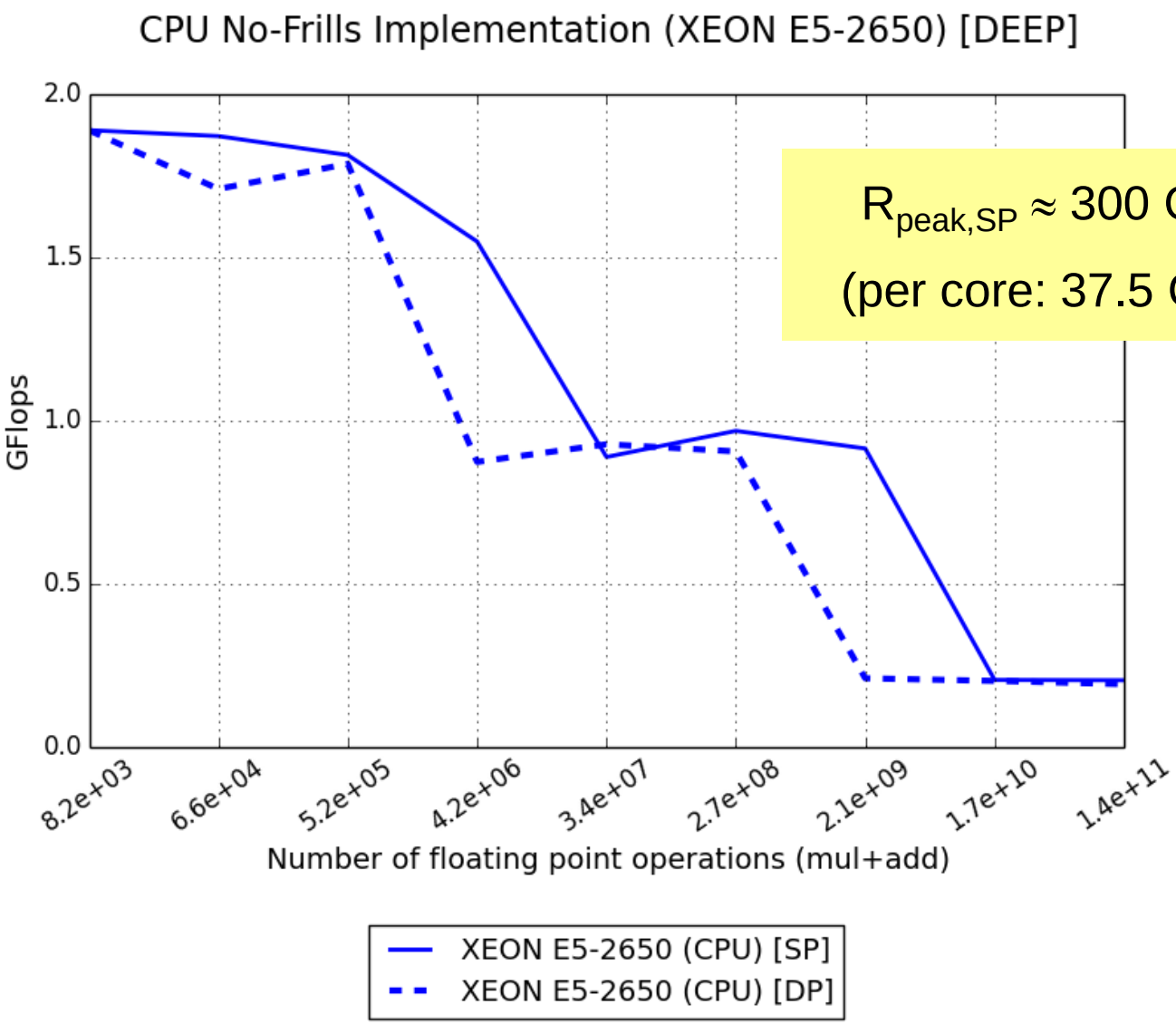
Simple C struct to hold matrix data:

```
// basic matrix data type
struct simpleMatrix {
  unsigned int width;
  unsigned int height;
  unsigned int stride;
  size_t bufferSize;
  real_t* ptr;
};
```

Matrix multiplication on the host:

```
// no-frills CPU implementation of matrix multiplication
void multCPU( simpleMatrix &C, const simpleMatrix &A,
              const simpleMatrix &B )
{
  for( int y=0; y < C.height; y++ ) {
    for( int x=0; x < C.width; x++ ) {
      real_t CVal = 0.0;
      for( int e=0; e < A.width; e++ ) {
        CVal += A.ptr[y*A.stride+e] * B.ptr[e*B.stride+x];
      }
      C.ptr[y*C.stride+x] = CVal;
    }
  }
  return;
}
```

Performance of “Naive” Host Code



$R_{peak,SP} \approx 300$ GFlops

(per core: 37.5 GFlops)

Exercise 1: Tasks

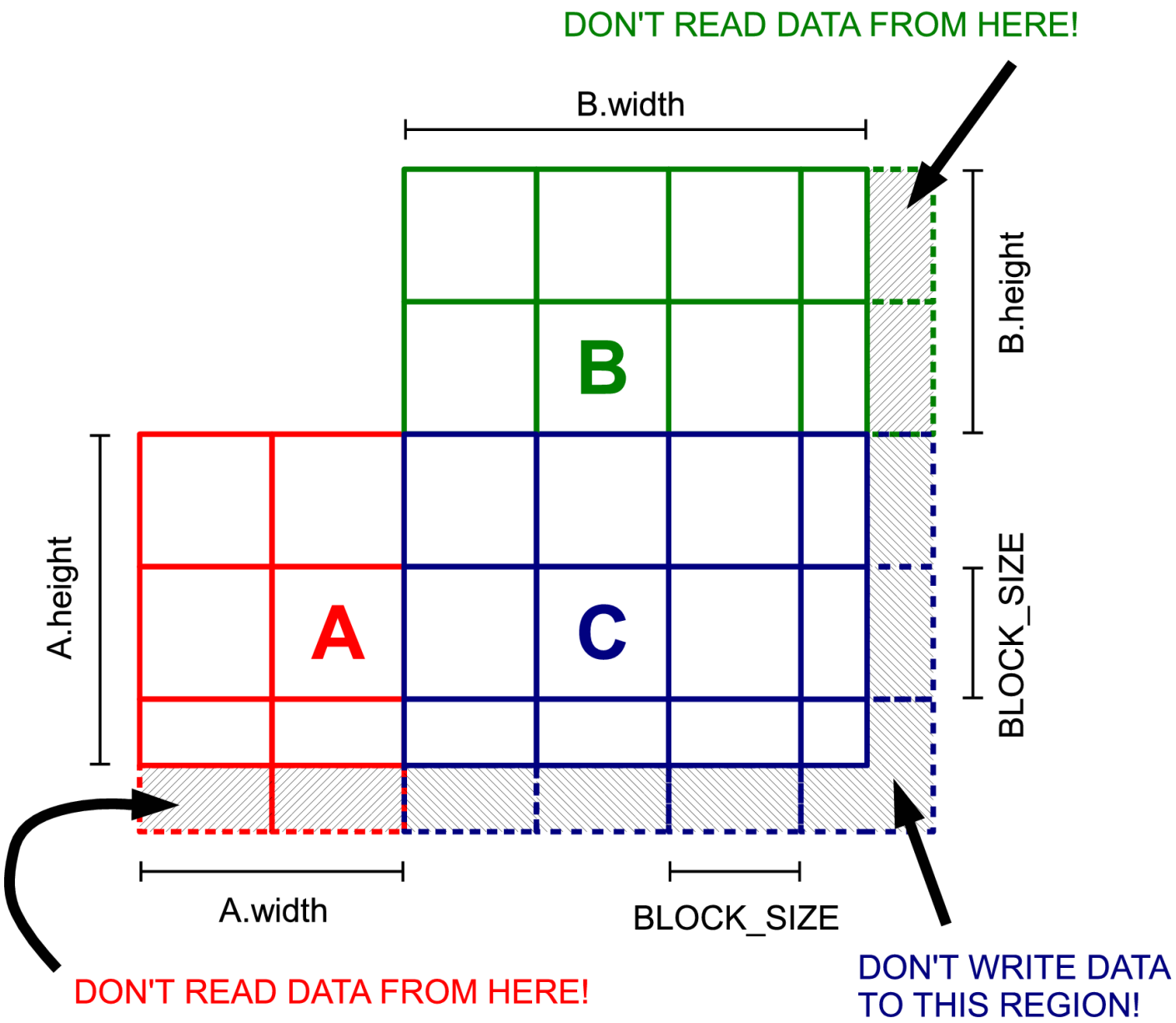
- Write your own matrix multiplication kernel: Every work-item shall compute a single element of C!
- Compare the result with the “gold standard” (computation on the host)!
- Do some benchmarking host vs. device on the system of your choice!
  (use for now a work-group size of 16x16)

Exercise 1: Hints

- Copy project files from `train060` account
- Adjust library and include paths in `Makefile` according to the machine you are using (just comment in and out)
- Helper files (no editing necessary):
  `matmul_helpers.[CH]`
  `opencl_helpers.[CH]`
- Host code to edit and modify (search for TODOs):
  `matmul_opencl.C`
- Device code to edit and modify (search for TODOs):
  `matmul.cl`

- Start by building the binary (invoke `make`) and by running it with the help option:
  `./MatMul -h`

Exercise 1: Hints (cont.)



Exercise 1: Solution

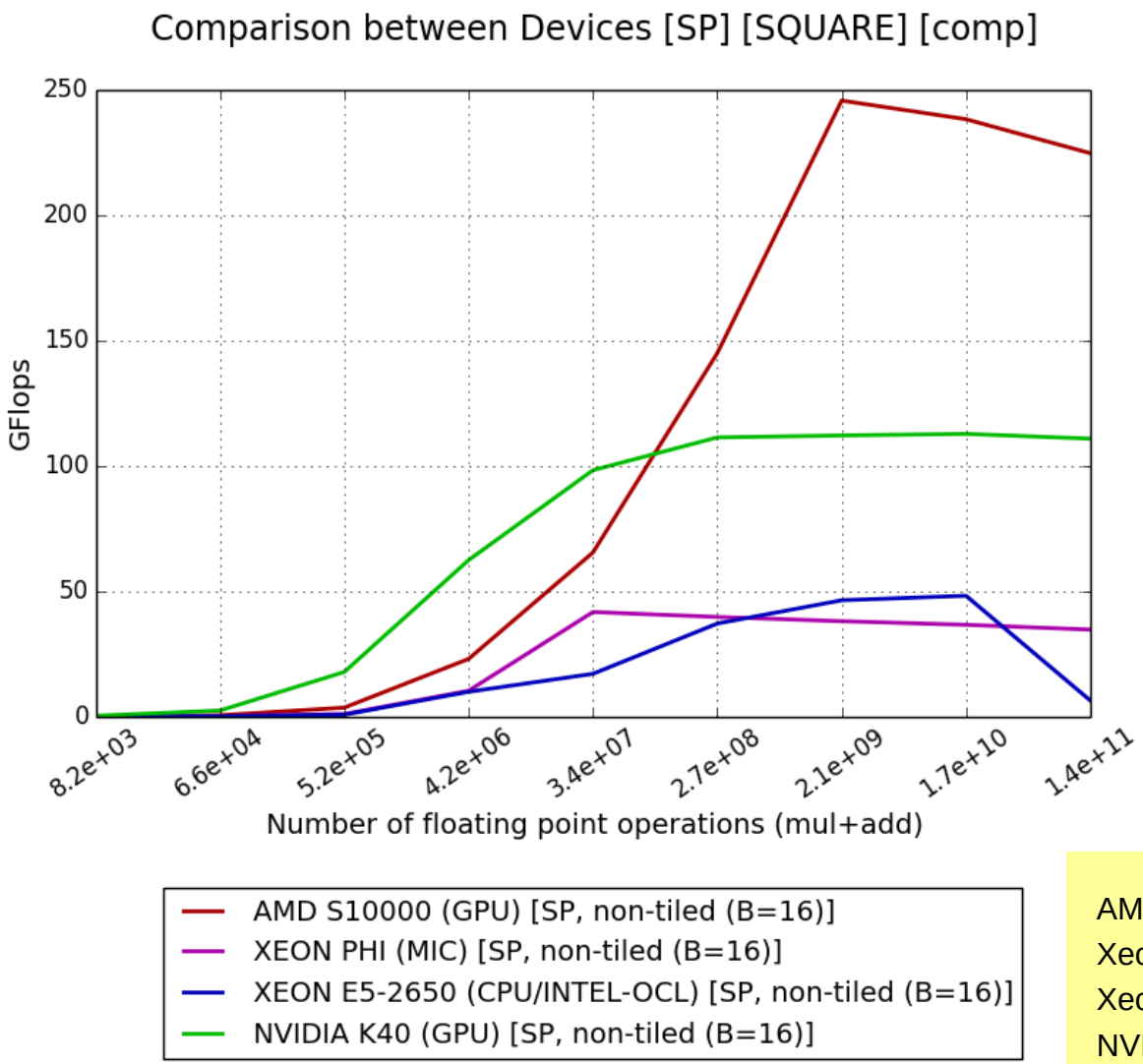
OpenCL kernel code:

```
__kernel __attribute__((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
         __attribute__((vec_type_hint(real_t)))

__kernel void matMulKernel(
              int Aheight, int Awidth,
              int Bheight, int Bwidth,
              int Astride, int Bstride, int Cstride,
              __global real_t* Aelements,
              __global real_t* Belements,
              __global real_t* Celements )
{
  // Get global indices of work-item
  int global_row = get_global_id(1);
  int global_col = get_global_id(0);

  // Check if we are within valid area of matrix C
  if( global_row < Aheight && global_col < Bwidth ) {
    // Compute single element of C
    real_t Cvalue = 0;
    for (int e = 0; e < Awidth; ++e)
      Cvalue += Aelements[global_row * Astride + e]
              * Belements[e * Bstride + global_col];
    // Write result into C matrix
    Celements[global_row * Cstride + global_col] = Cvalue;
  }
}
```

Kernel function qualifiers: Hints for OpenCL compiler

Performance Comparison between Devices

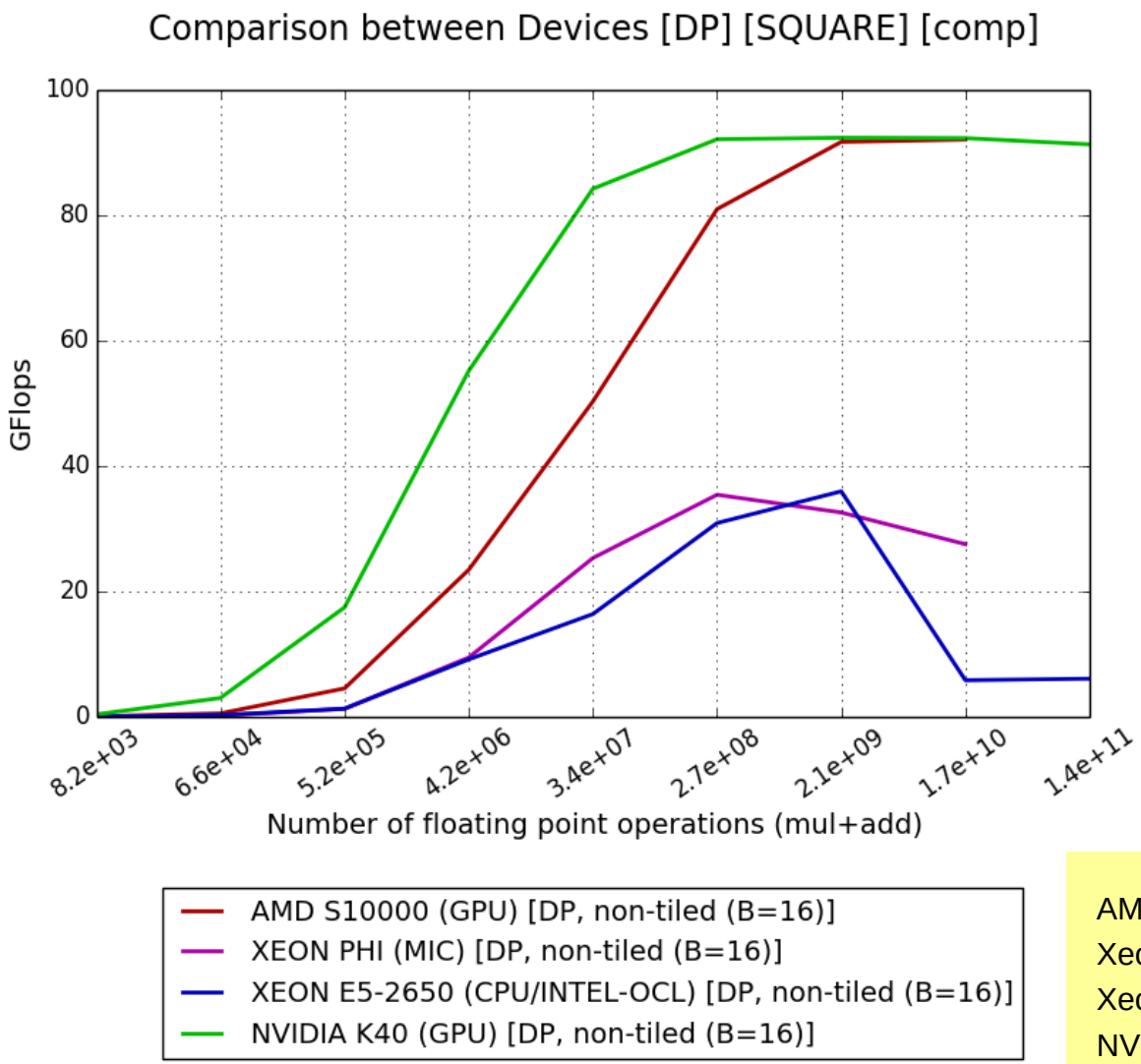
Simple Matrix Multiplication (SP)



AMD S10000:	$R_{peak,SP} \approx 2900$ GFlops
Xeon Phi:	$R_{peak,SP} \approx 2000$ GFlops
Xeon E5-2650 (2x):	$R_{peak,SP} \approx 600$ GFlops
NVIDIA K40:	$R_{peak,SP} \approx 5000$ GFlops

Performance Comparison between Devices

Simple Matrix Multiplication (DP)

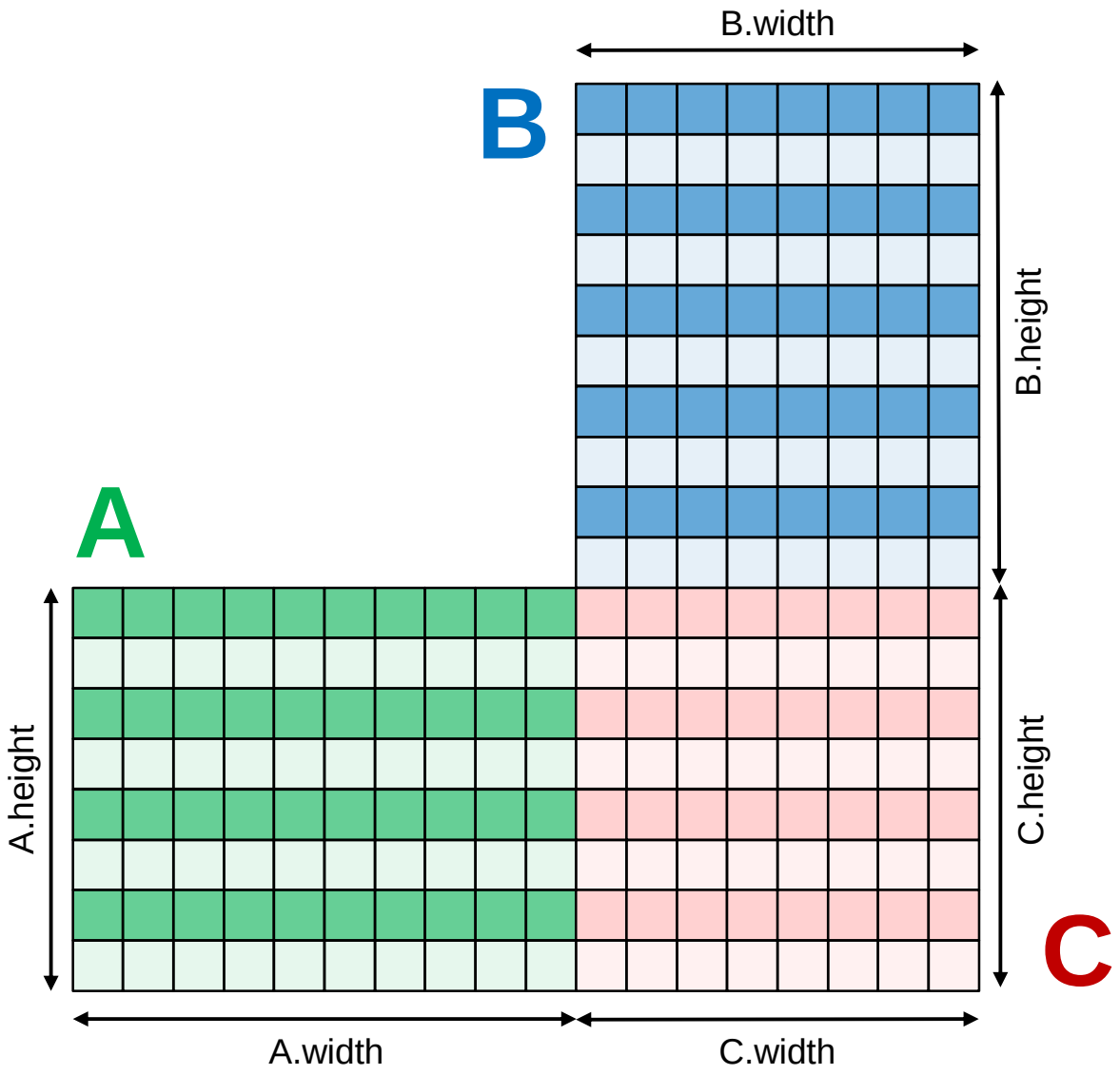


AMD S10000:	$R_{peak,DP} \approx 730$ GFlops
Xeon Phi:	$R_{peak,DP} \approx 1000$ GFlops
Xeon E5-2650 (2x):	$R_{peak,DP} \approx 300$ GFlops
NVIDIA K40:	$R_{peak,DP} \approx 1660$ GFlops

Memory Access Patterns

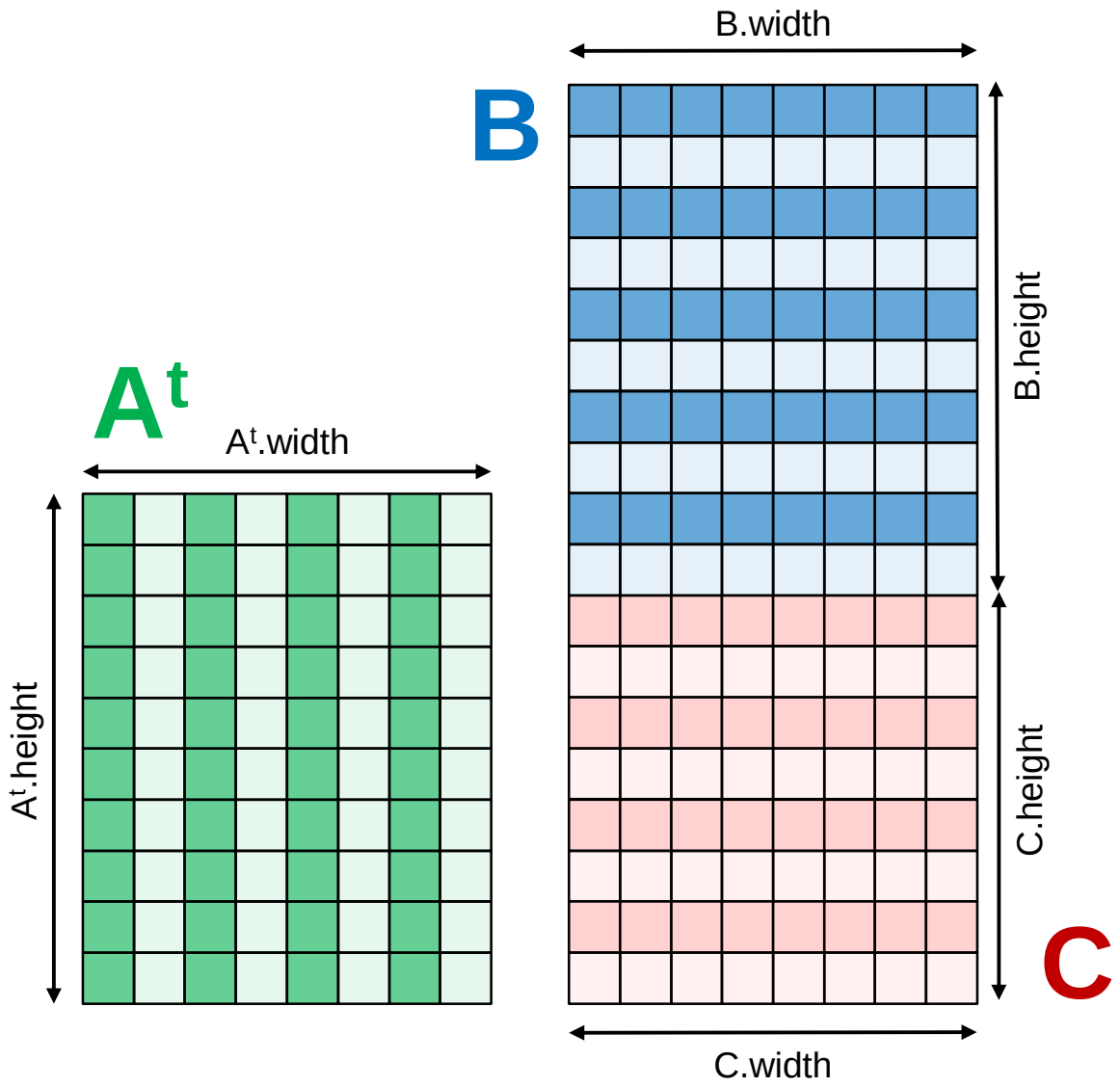
Varying the Memory Layout

Standard



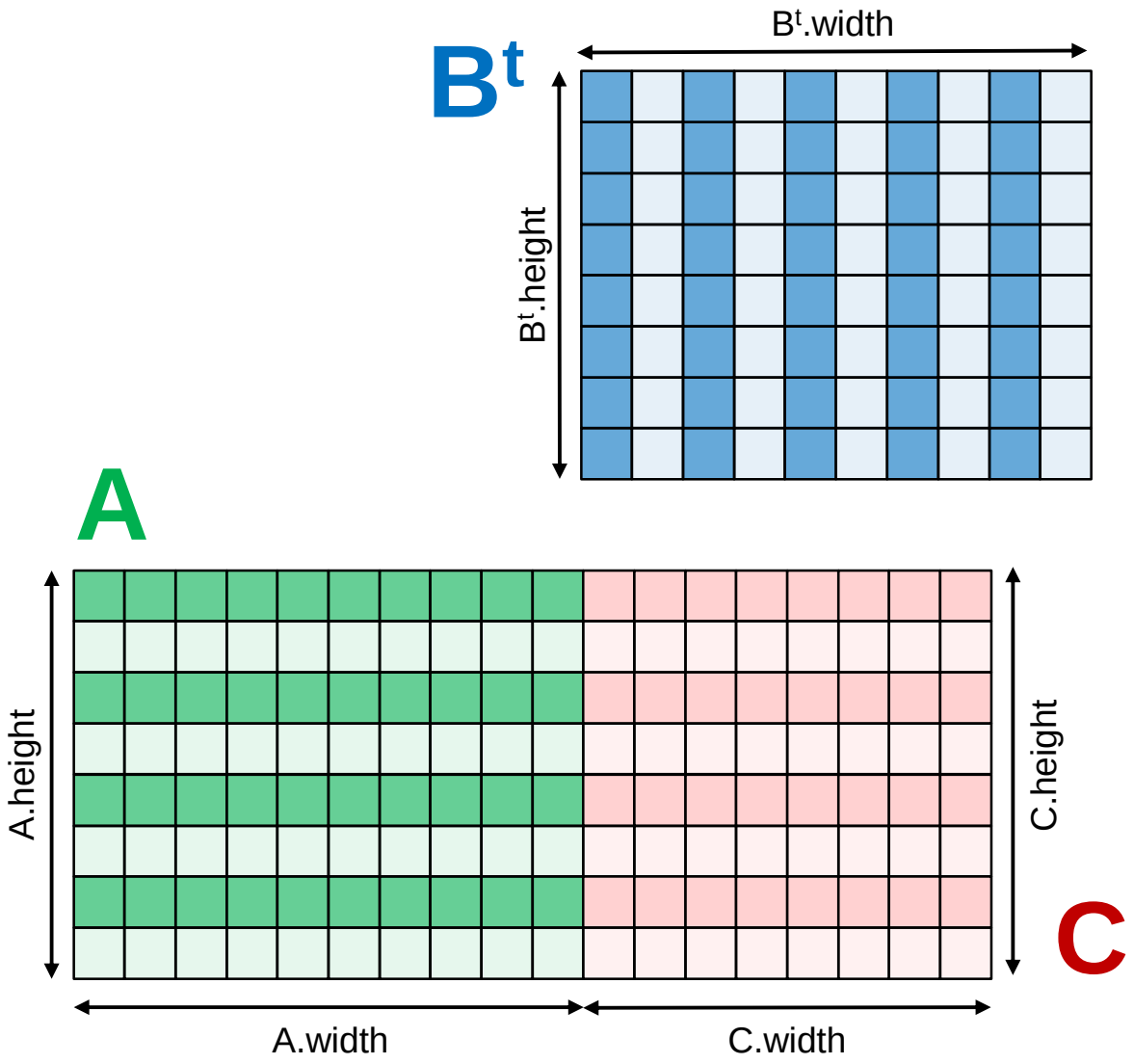
Varying the Memory Layout

Transposing A



Varying the Memory Layout

Transposing B



Exercise 2: Tasks and Hints

- Implement both kernel variations (**A** transposed and **B** transposed)!
- Compare the result with the “gold standard” (computation on the host)!
- Do some benchmarking of the different kernel variations on the system of your choice!

- Host code to edit and modify (search for TODOs): `matmul_opencl.C`
- Device code to edit and modify (search for TODOs): `matmul.cl`

Exercise 2: Solution for A

OpenCL kernel code:

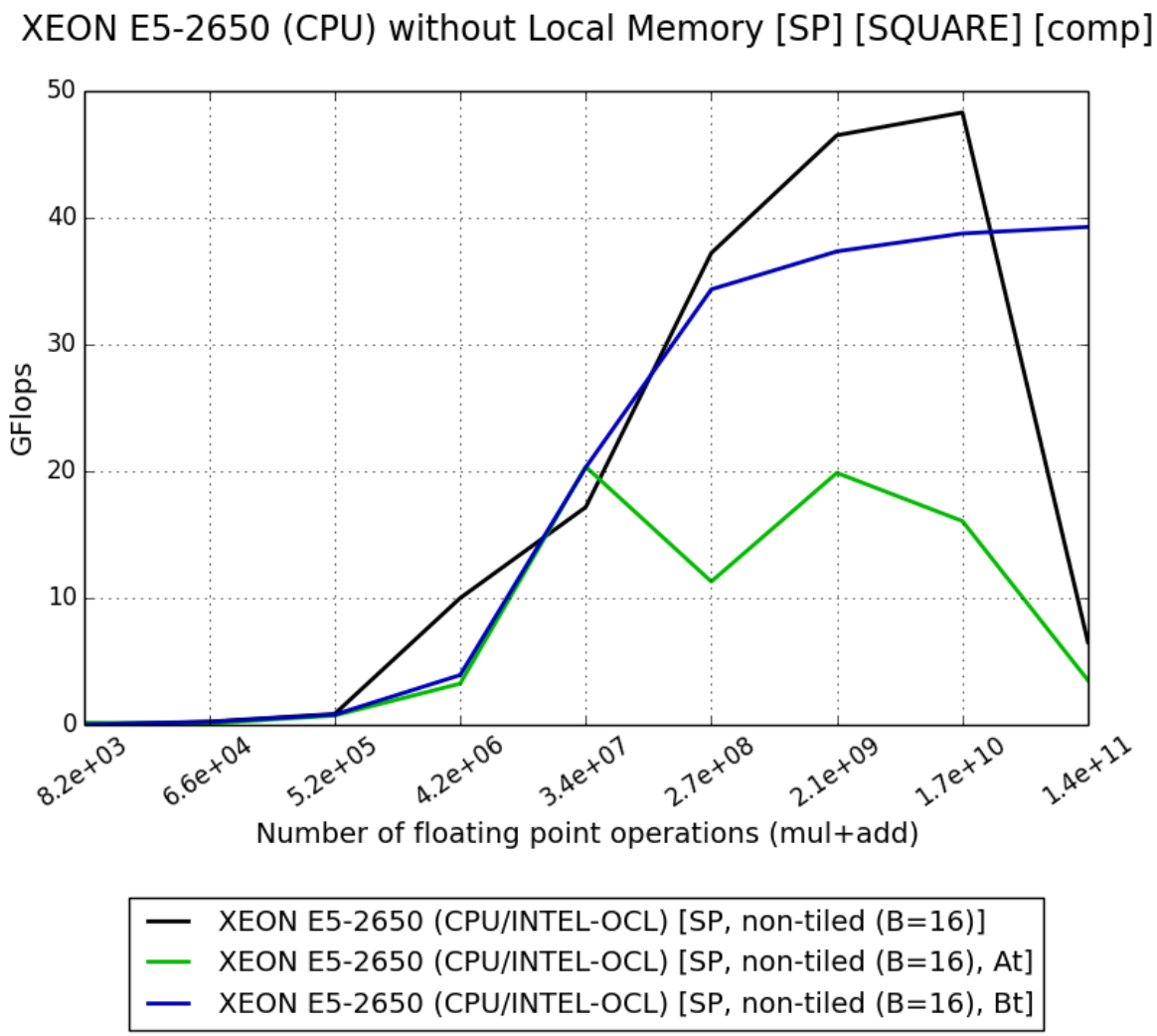
```
__kernel void matMulKernel_TRA(
              int Aheight, int Awidth,
              int Bheight, int Bwidth,
              int Astride, int Bstride, int Cstride,
              __global real_t* Aelements,
              __global real_t* Belements,
              __global real_t* Celements )
{
  // Get global indices of work-item
  int global_row = get_global_id(1);
  int global_col = get_global_id(0);

  // Check if we are within valid area of matrix C
  if( global_row < Awidth && global_col < Bwidth ) {
    // Compute single element of C
    real_t Cvalue = 0;
    for (int e = 0; e < Aheight; ++e)
      Cvalue += Aelements[e * Astride + global_row]
              * Belements[e * Bstride + global_col];
    // Write result into C matrix
    Celements[global_row * Cstride + global_col] = Cvalue;
  }
}
```

Exercise 2: Solution for B

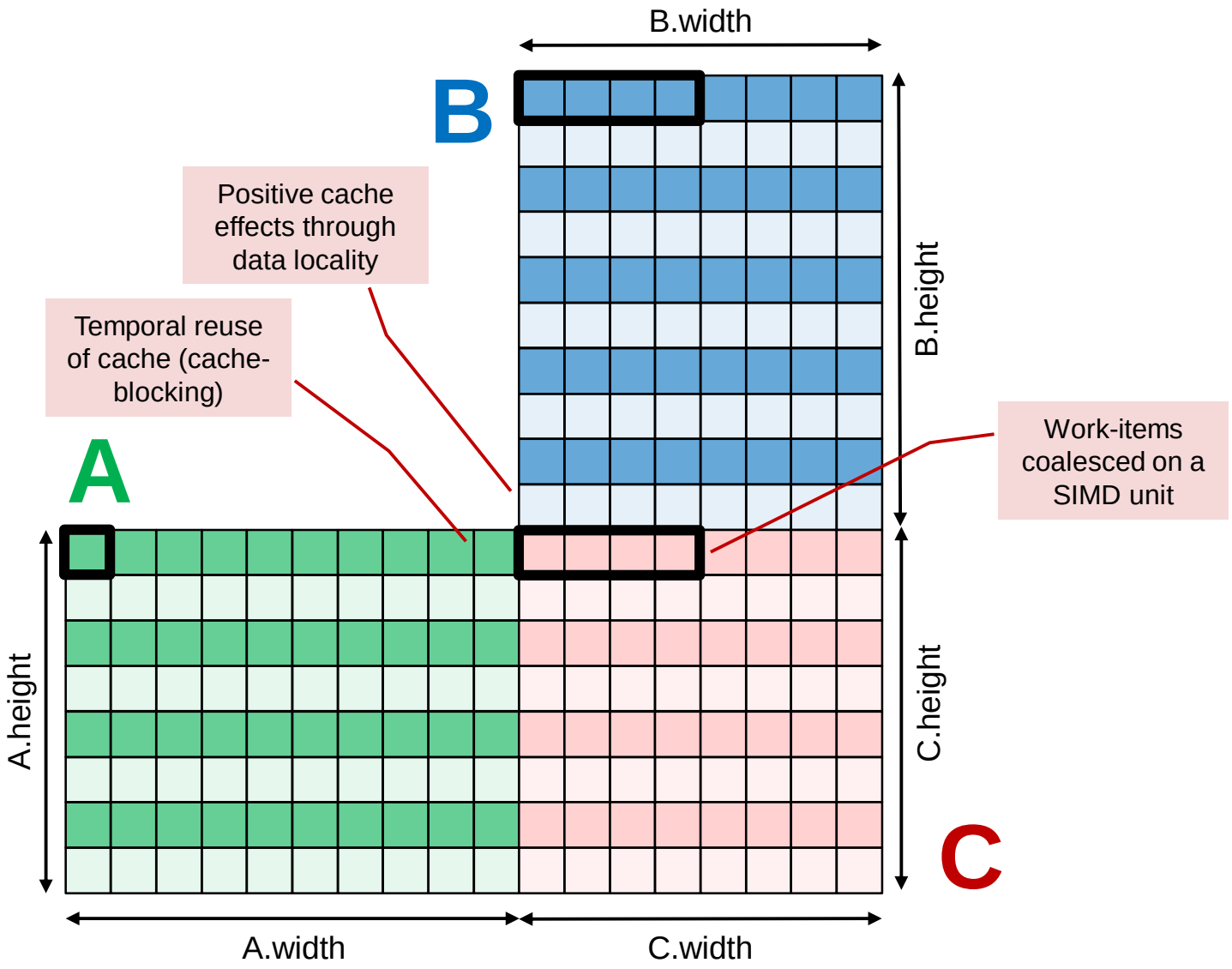
OpenCL kernel code:

```
__kernel void matMulKernel_TRB(
              int Aheight, int Awidth,
              int Bheight, int Bwidth,
              int Astride, int Bstride, int Cstride,
              __global real_t* Aelements,
              __global real_t* Belements,
              __global real_t* Celements )
{
  // Get global indices of work-item
  int global_row = get_global_id(1);
  int global_col = get_global_id(0);

  // Check if we are within valid area of matrix C
  if( global_row < Aheight && global_col < Bheight ) {
    // Compute single element of C
    real_t Cvalue = 0;
    for (int e = 0; e < Awidth; ++e)
      Cvalue += Aelements[global_row * Astride + e]
              * Belements[global_col * Bstride + e];
    // Write result into C matrix
    Celements[global_row * Cstride + global_col] = Cvalue;
  }
}
```

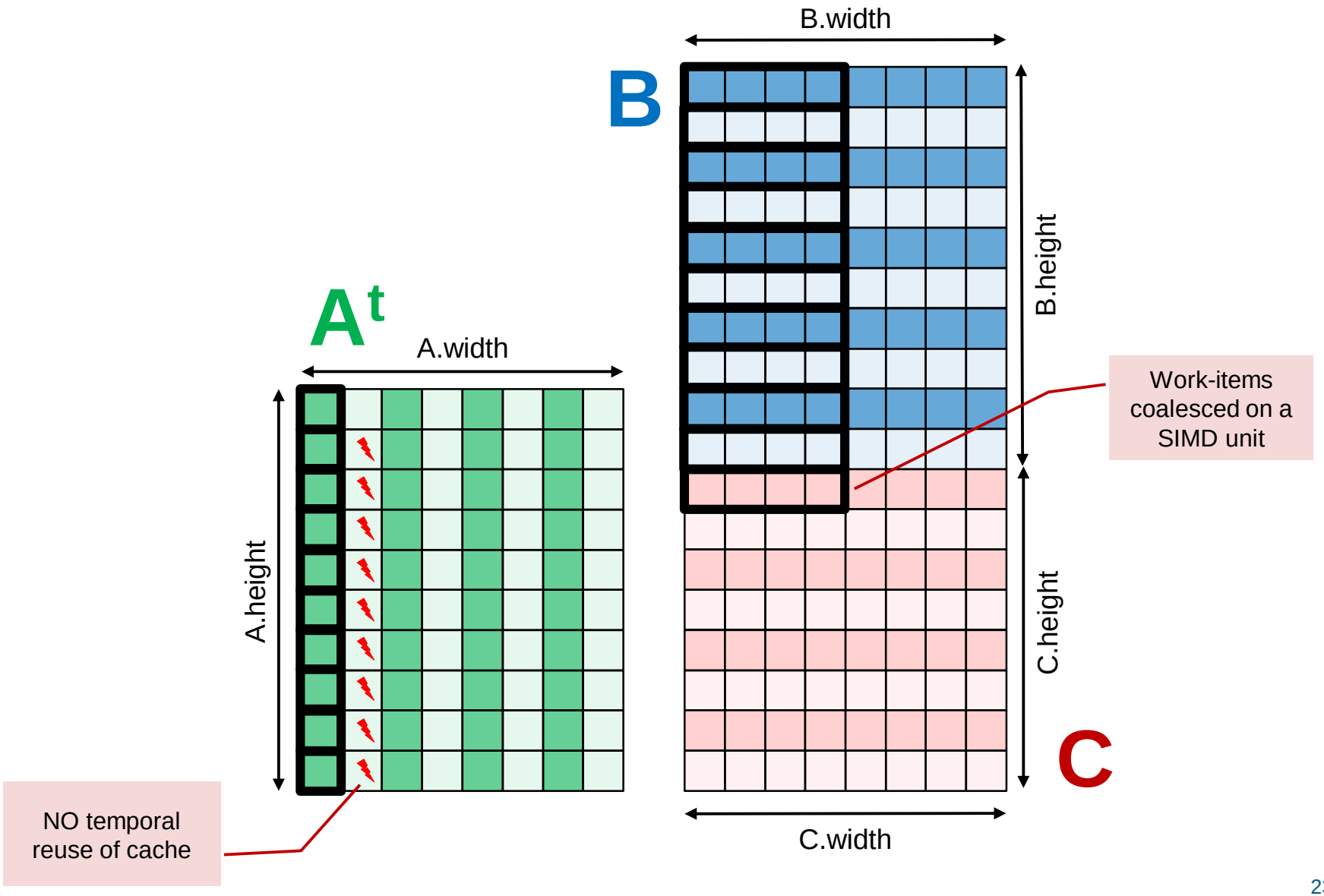
Results on the CPU
(Exercise 2)



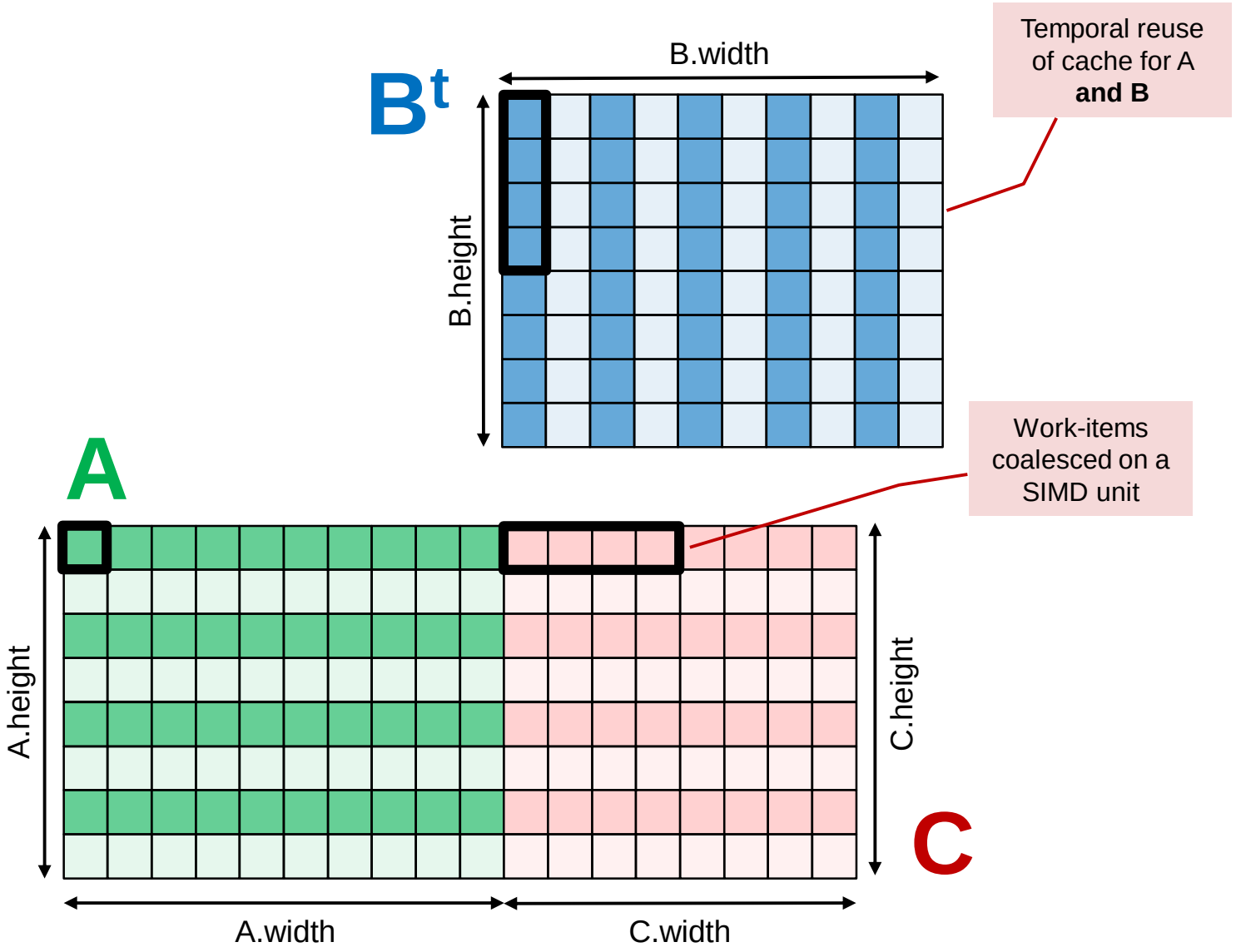
Memory Access on the CPU



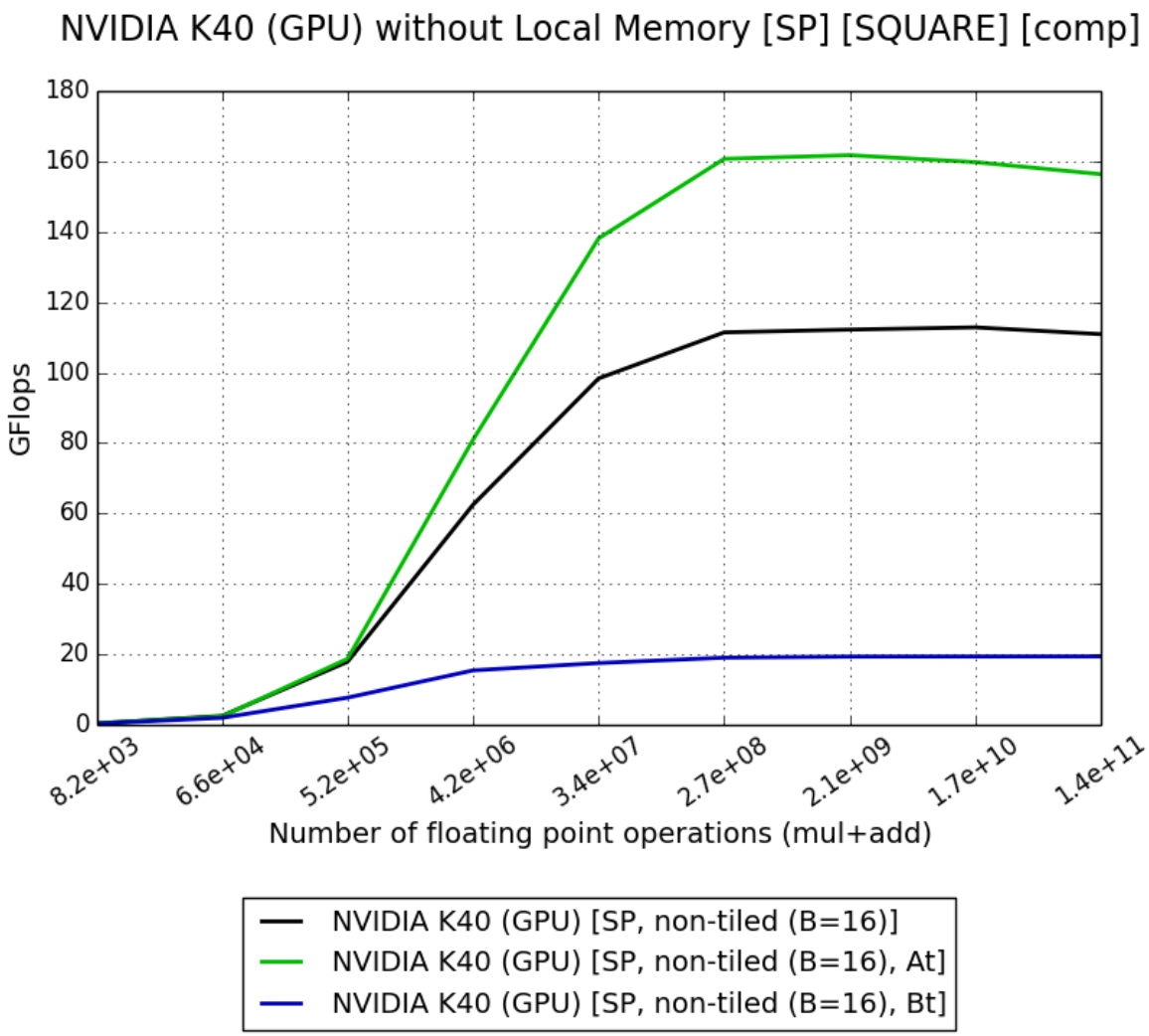
Transposing A on the CPU



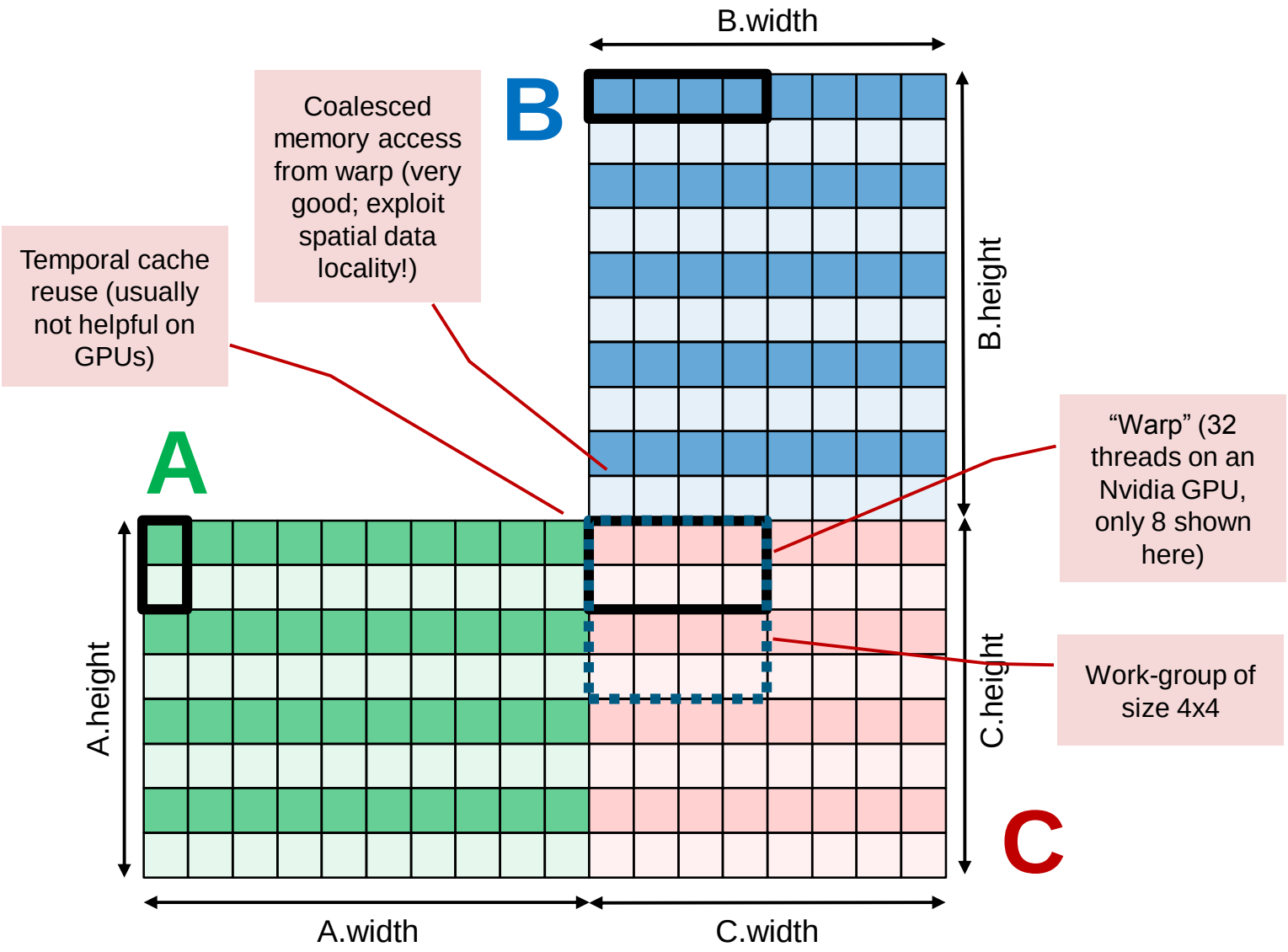
Transposing B on the CPU



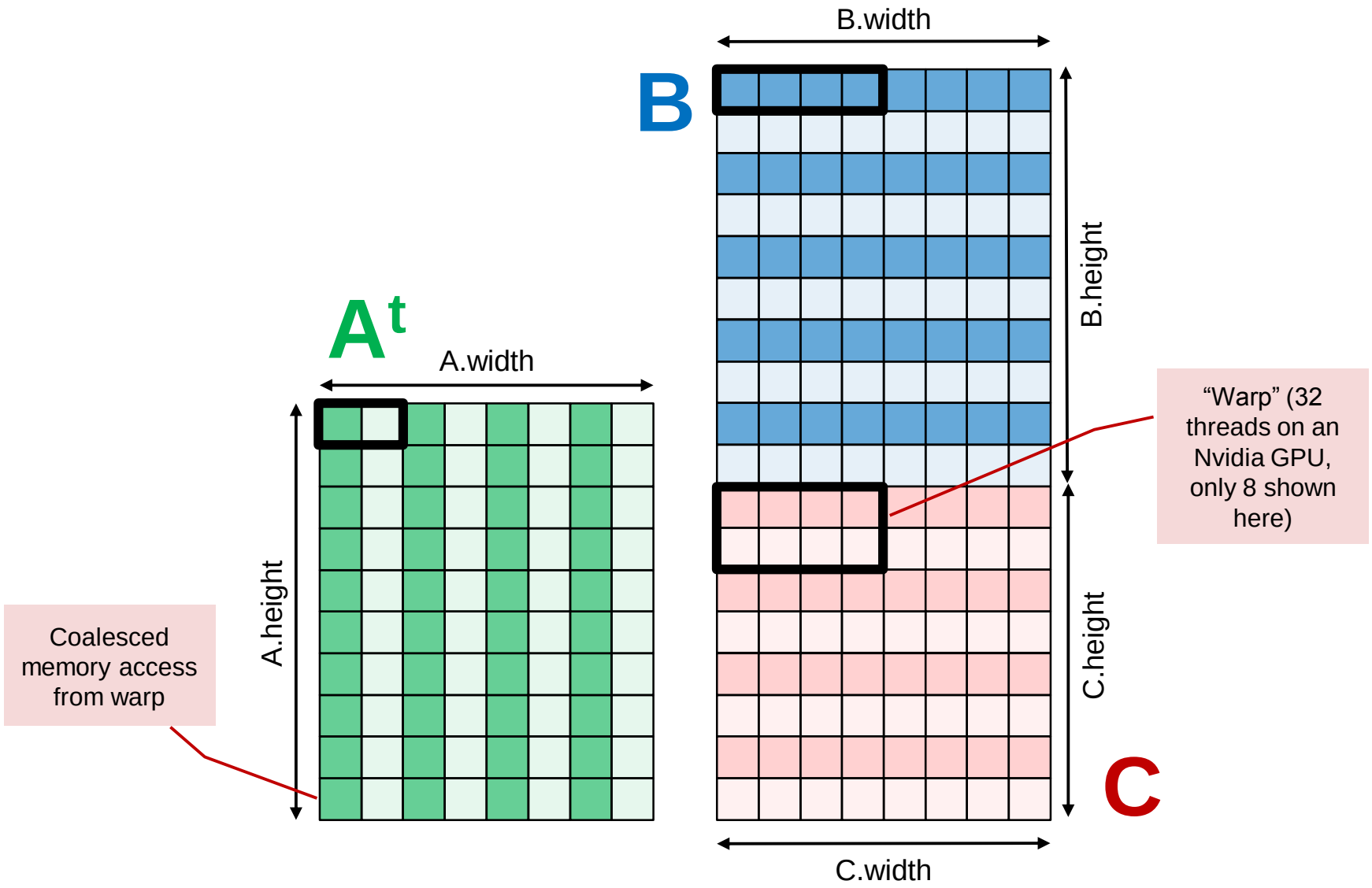
Results on the Nvidia GPU
(Exercise 2)



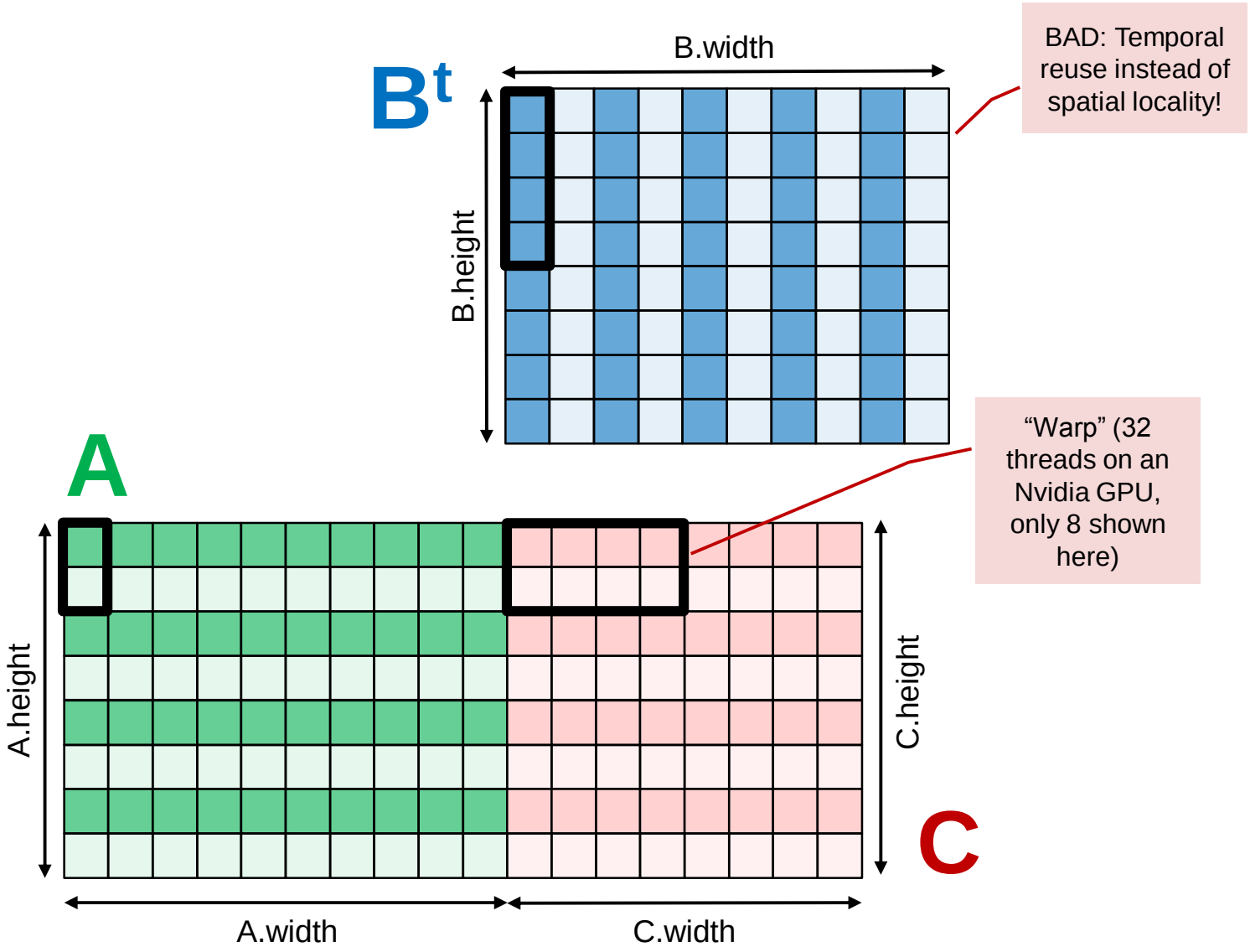
Memory Access on the GPU



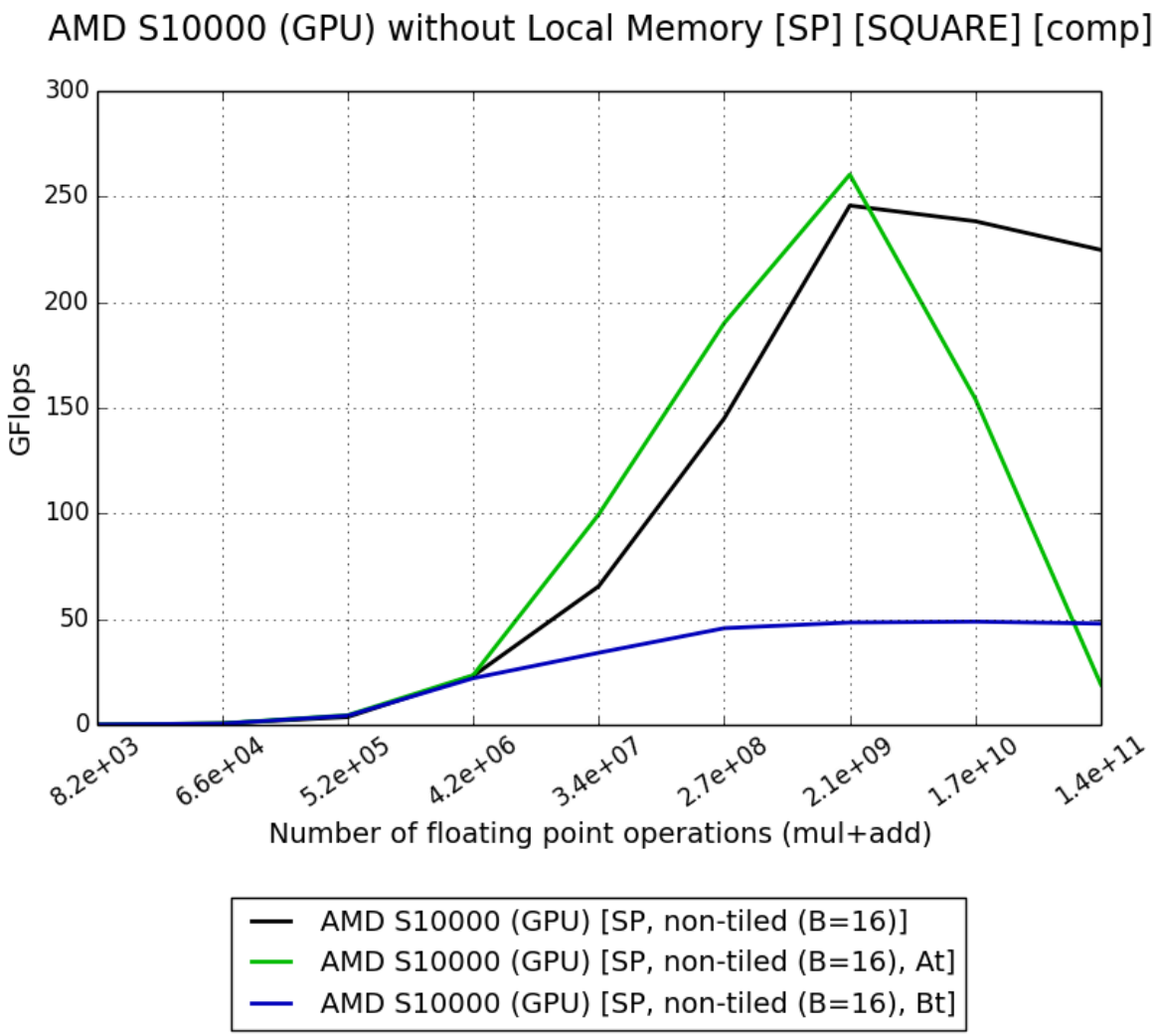
Transposing A on the GPU



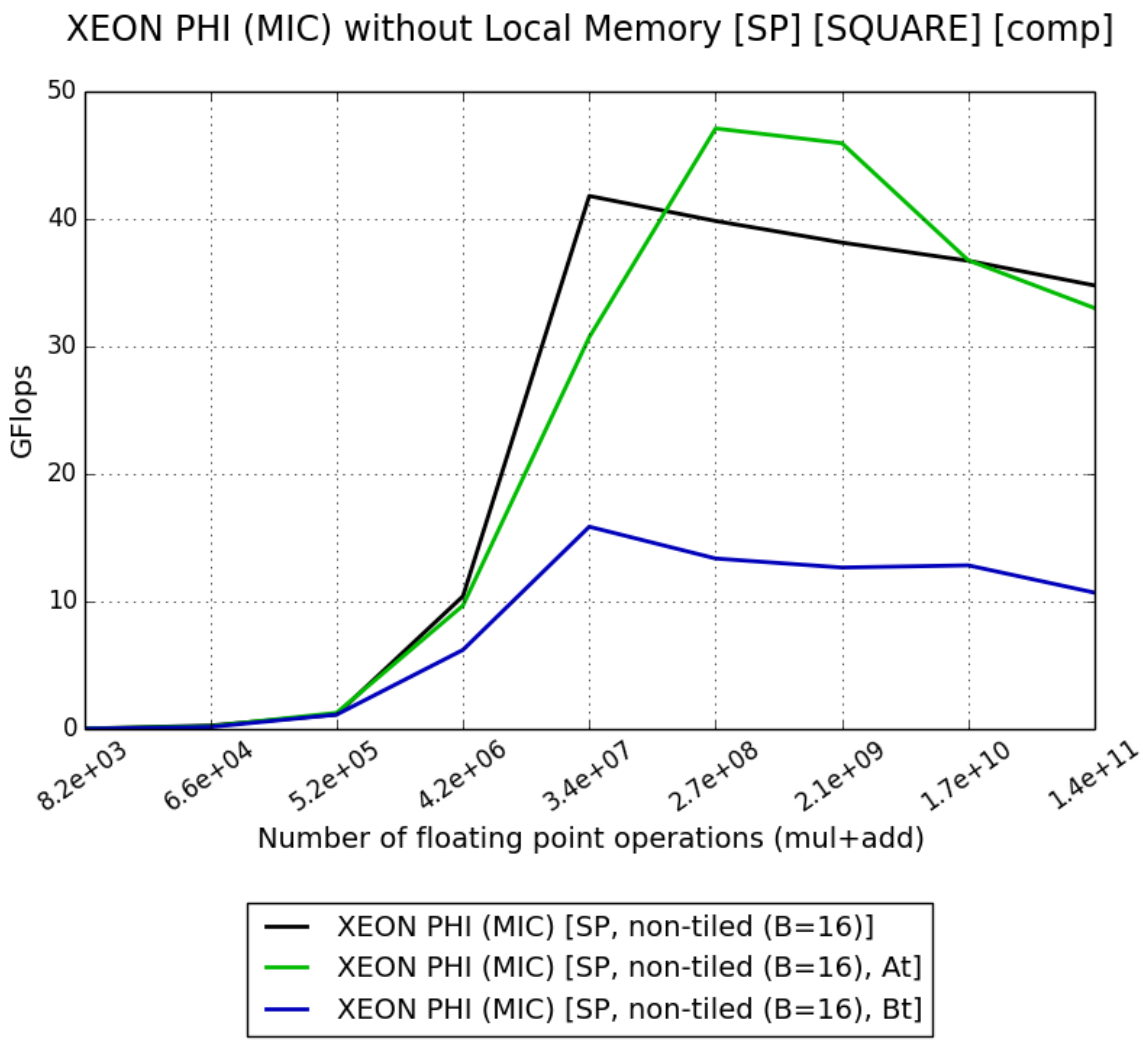
Transposing B on the GPU



Results on the AMD GPU
(Exercise 2)



Results on Xeon Phi
(Exercise 2)



Summary: Memory Access

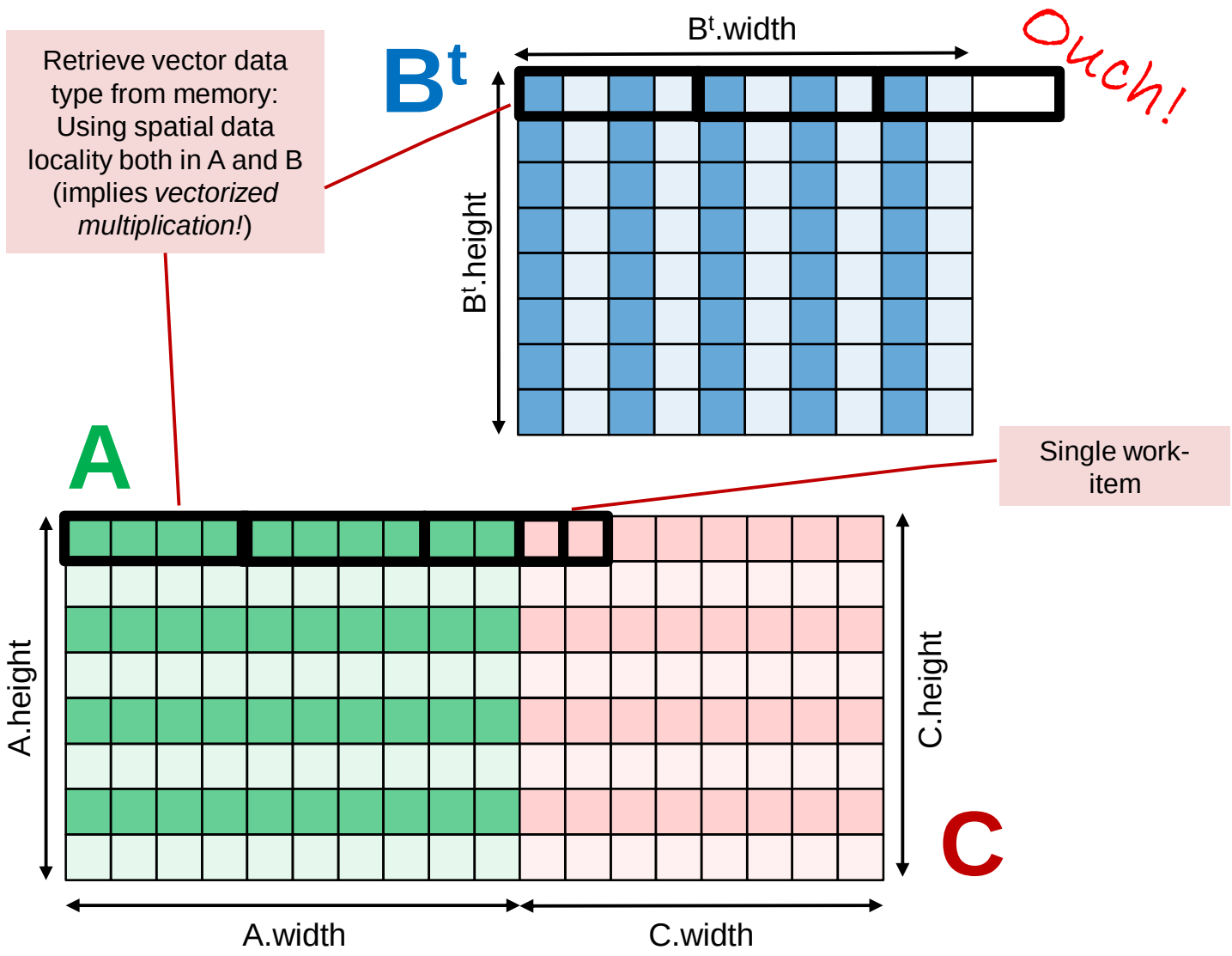
- CPUs love temporal reuse of cache contents
- GPUs love (mostly) spatial data locality (coalesced memory access from all work-items within a warp/wavefront)
- MICs behave similar to GPUs in this study (surprisingly!)

Manual Vectorization (esp. for CPUs)

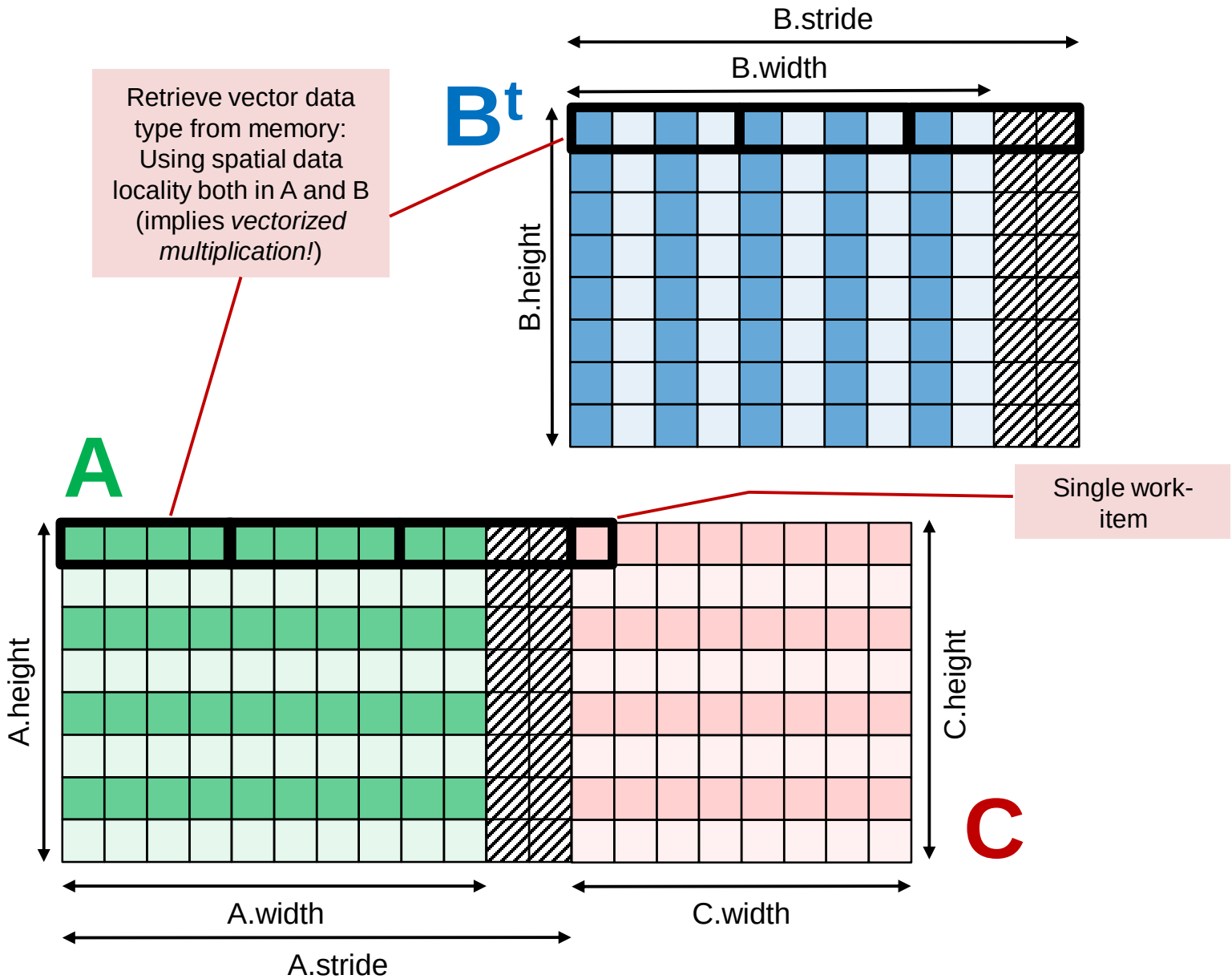
Manual Vectorization

- Use OpenCL vector data types in compute kernel

Approach on Basis of Transposed B Matrix



Approach on Basis of Transposed B Matrix
(Arrays with stride)



Manual Vectorization (cont.)

- Use OpenCL vector data types in compute kernel
- The stride of **A** and $\mathbf{B}^t$ has to be a multiple of the vector length (called `alignmentDesired` in the matrix multiplication host program)

Exercise 3: Tasks and Hints

- Implement a kernel variation for a transposed **B** matrix with vectorized access to the elements of A and B!
- Compare the result with the “gold standard”!
- Do some benchmarking!

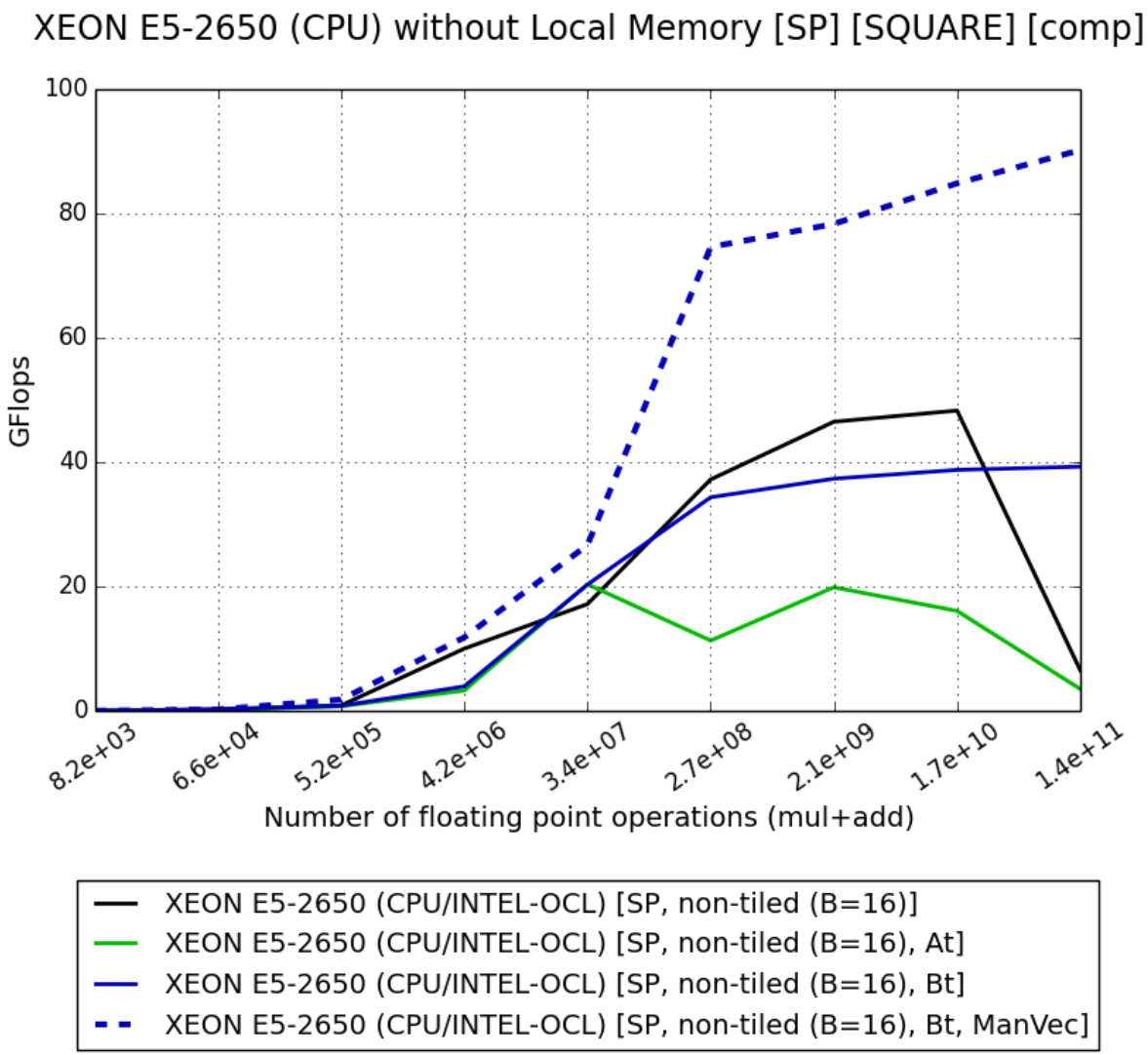
- Host code to edit and modify (search for TODOs): `matmul_opencl.C`
- Device code to edit and modify (search for TODOs): `matmul.cl`
- <u>For simplification:</u> Develop a solution just for `float` as basic data type, assume a vector length of 8 (see `alignmentDesired` in `matmul_opencl.C`) (therefore `float8` is the way to go, stride of the arrays is already correct for this setting).

Exercise 3: Solution

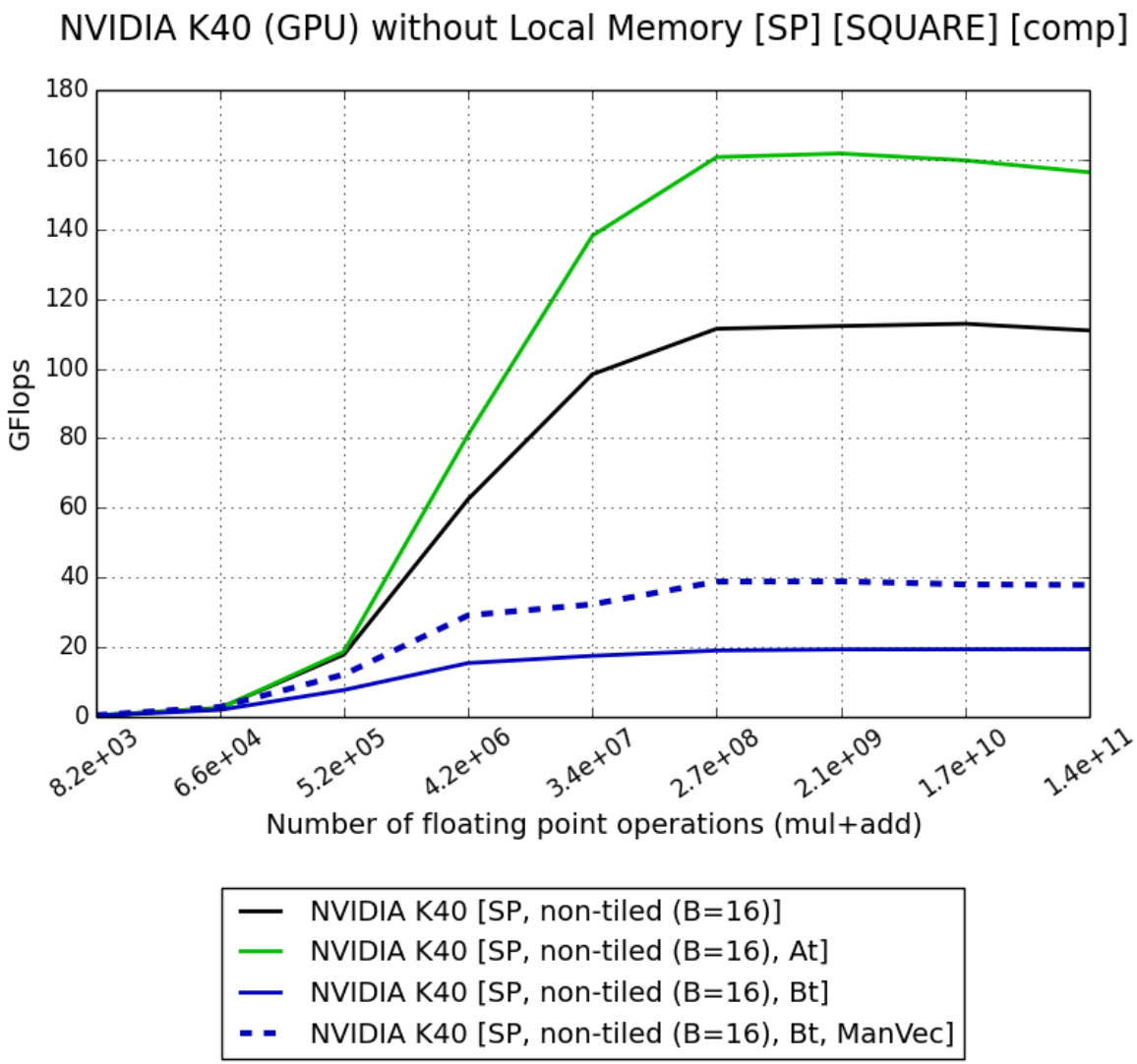
OpenCL kernel code

```
__kernel void matMulKernel_TRB_VEC(
              int Aheight, int Awidth,
              int Bheight, int Bwidth,
              int Astride, int Bstride, int Cstride,
              __global float8* Aelements,
              __global float8* Belements,
              __global float* Celements )
{
  // Get global indices of work-item
  int global_row = get_global_id(1);
  int global_col = get_global_id(0);

  // Account for vector length of 8 in width of A and stride of A and B
  uint Awidth8 = (Awidth - 1) / 8 + 1;
  uint Astride8 = Astride / 8;
  uint Bstride8 = Bstride / 8;

  float8 Aelems, Belems, Celems;
  // Check if we are within valid area of matrix C
  if( global_row < Aheight && global_col < Bheight ) {
    // Compute single element of C
    real_t Cvalue = 0;
    for (int e = 0; e < Awidth8; e++) {
      Aelems = Aelements[global_row * Astride8 + e];
      Belems = Belements[global_col * Bstride8 + e];
      Celems = Aelems * Belems;

      Cvalue += Celems.s0;
      Cvalue += Celems.s1;
      Cvalue += Celems.s2;
      Cvalue += Celems.s3;
      Cvalue += Celems.s4;
      Cvalue += Celems.s5;
      Cvalue += Celems.s6;
      Cvalue += Celems.s7;
    }
    // Write result into C matrix
    Celements[global_row * Cstride + global_col] = Cvalue;
  }
}
```

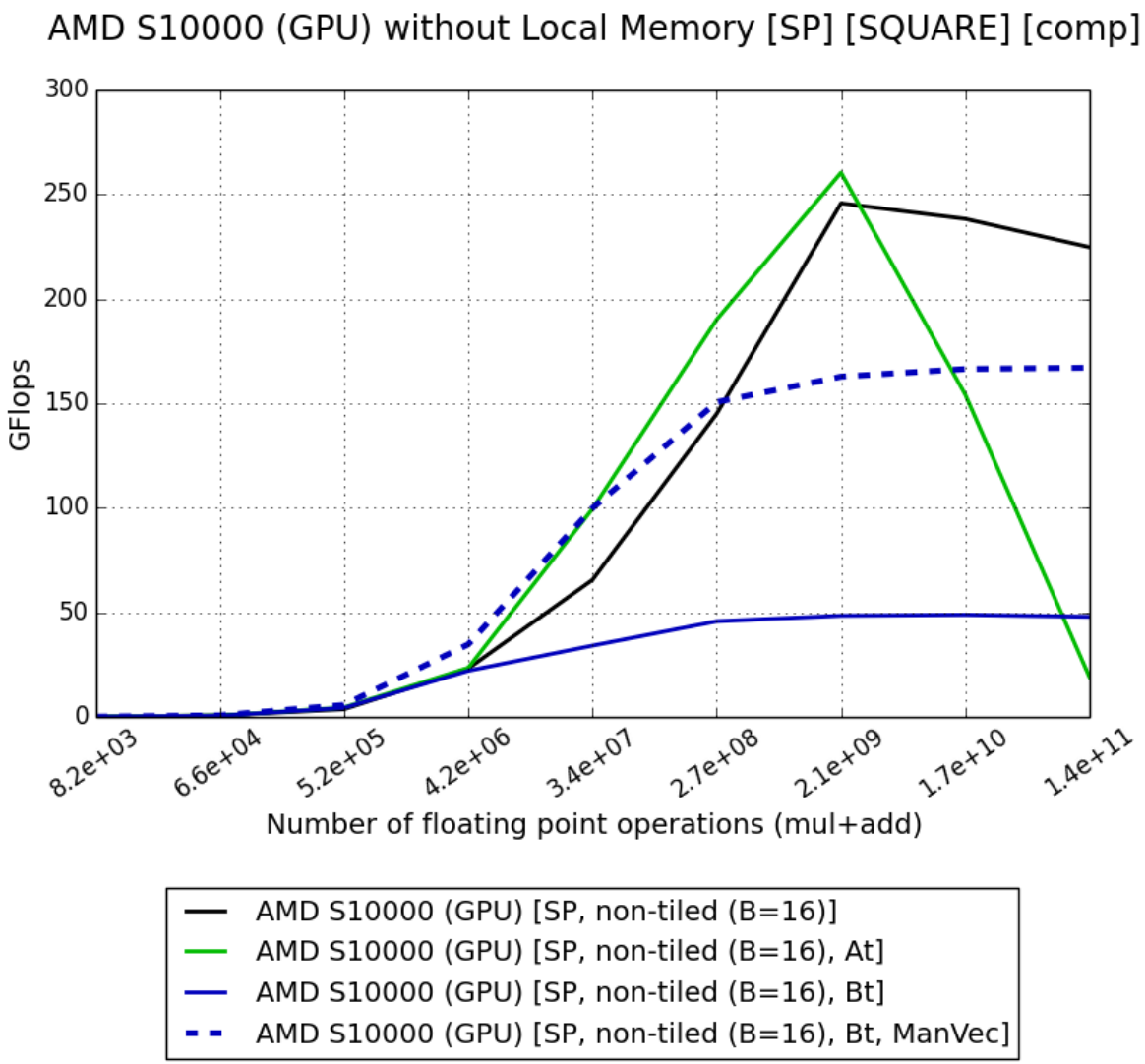
Results on the CPU
(Exercise 3)



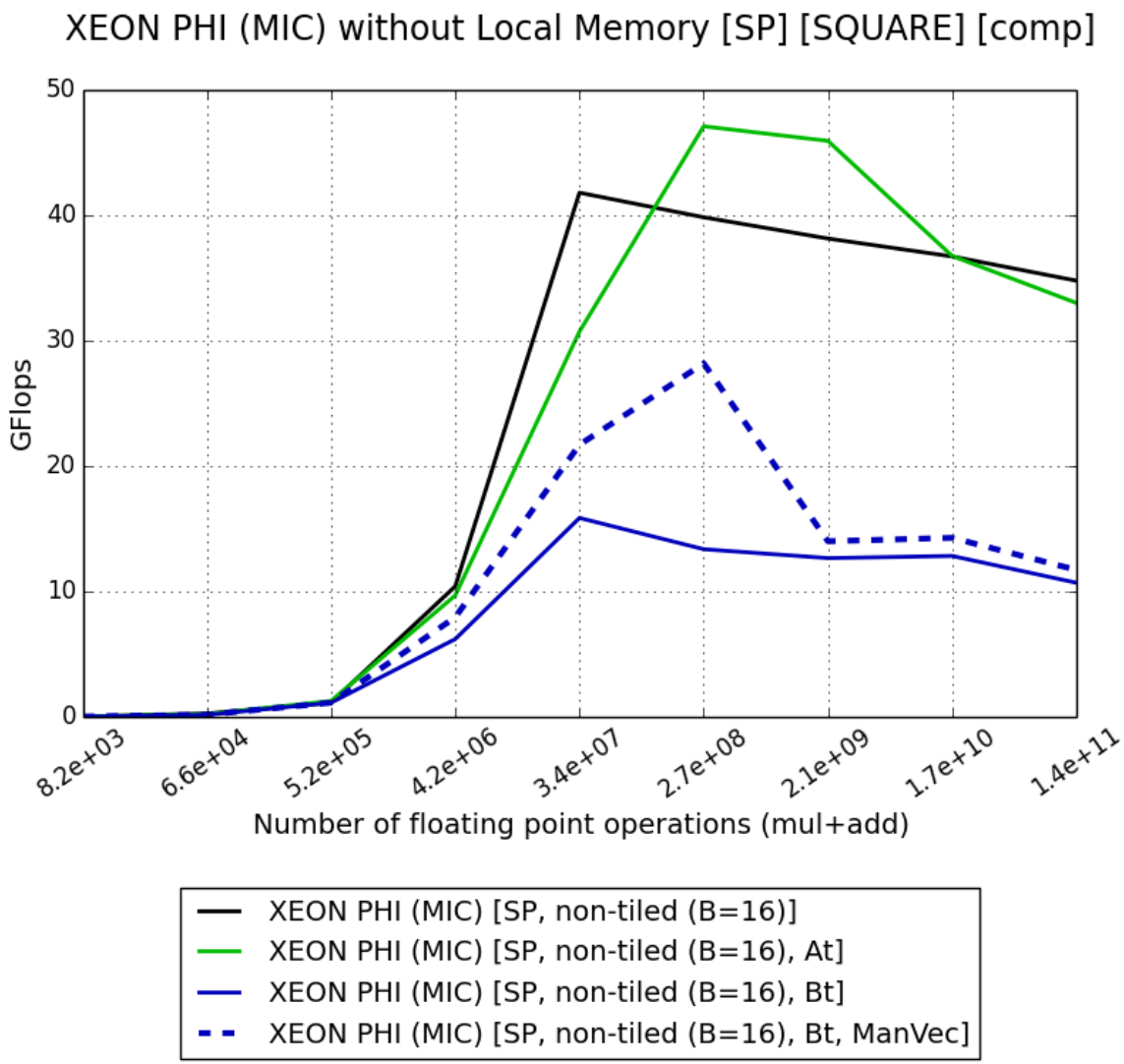
Results on the Nvidia GPU
(Exercise 3)



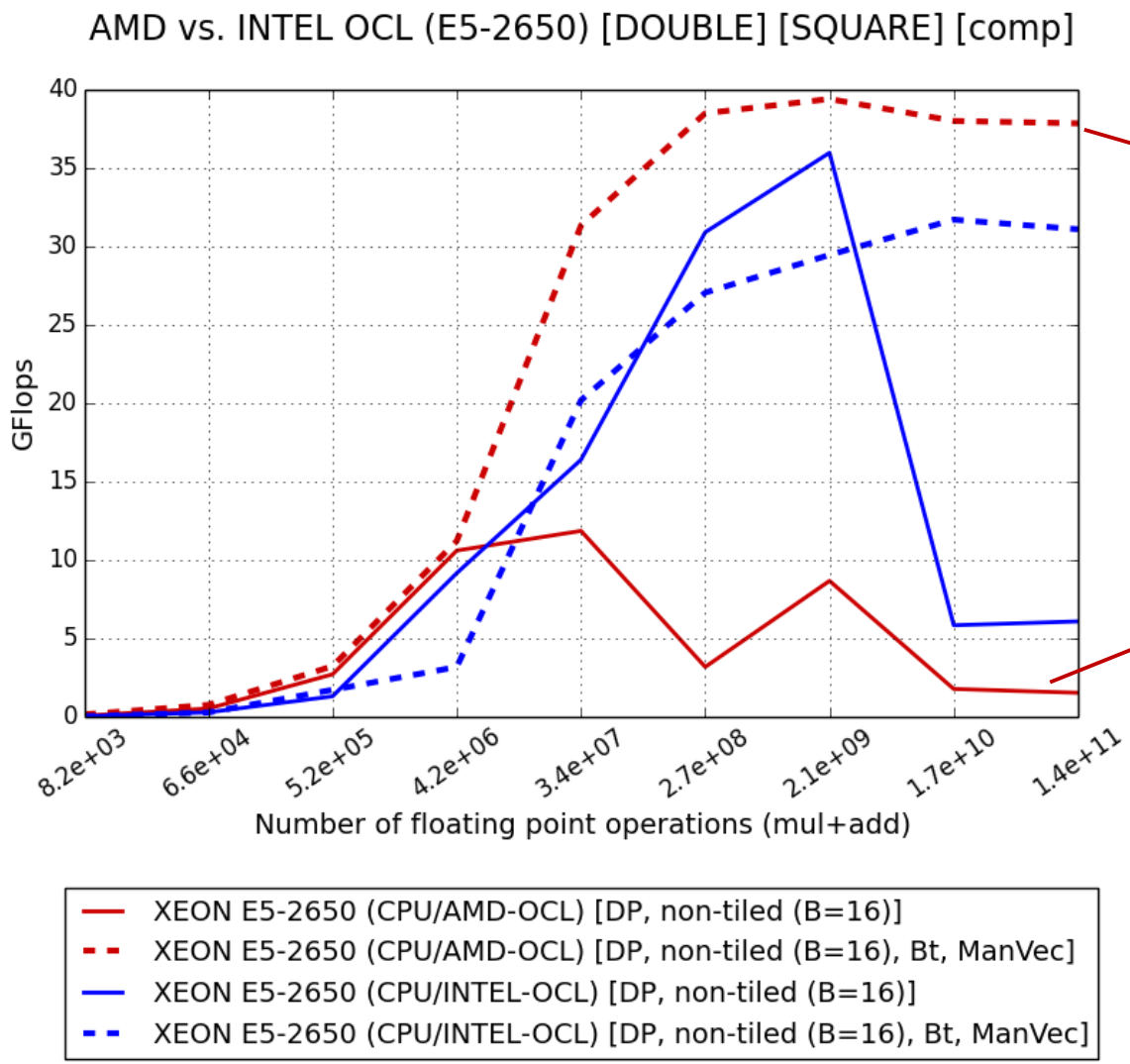
Results on the AMD GPU
(Exercise 3)



Results on Xeon Phi
(Exercise 3)



Comparing CPU Platforms (AMD vs. Intel)



AMD OCL compiler handles manual vectorization very well

AMD OCL compiler without auto-vectorization

Summary: Manual Vectorization

- Significant improvement for CPU device
  - especially on AMD OpenCL platform
- Better than just using transposed **B** matrix on all devices
- However: Not the best solution for GPUs or MIC

Using Local Memory

Using Local Memory

- **Motivation:**
  - Local memory can be used as programmer-managed cache (only on GPUs)
  - Reduces number of accesses to global memory

Matrix Multiplication with Local Memory

Overview

Basic idea: Every work-group is responsible for a different submatrix within **C**

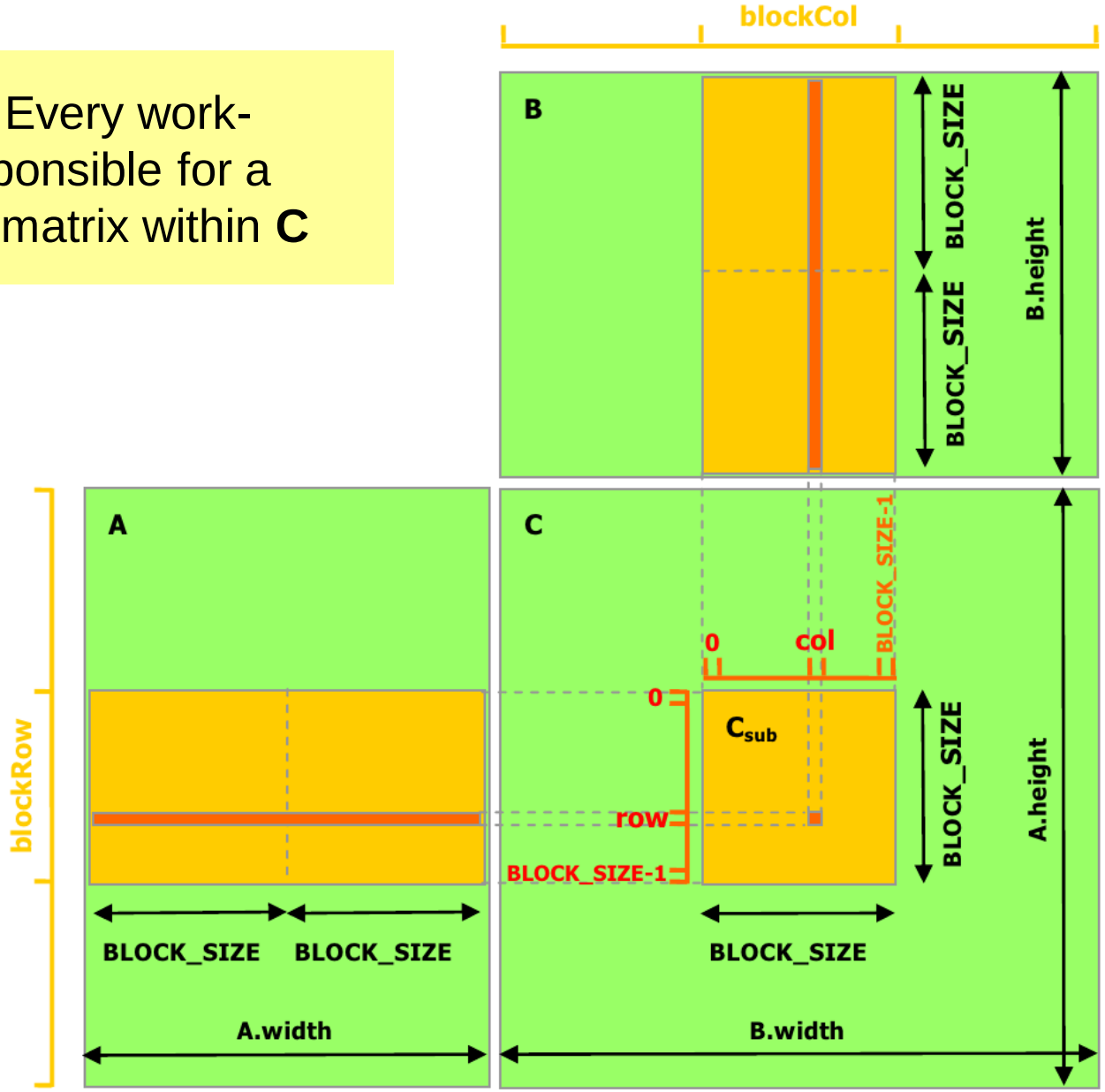


Fig.: NVIDIA

Matrix Multiplication with Local Memory

Basic Algorithm

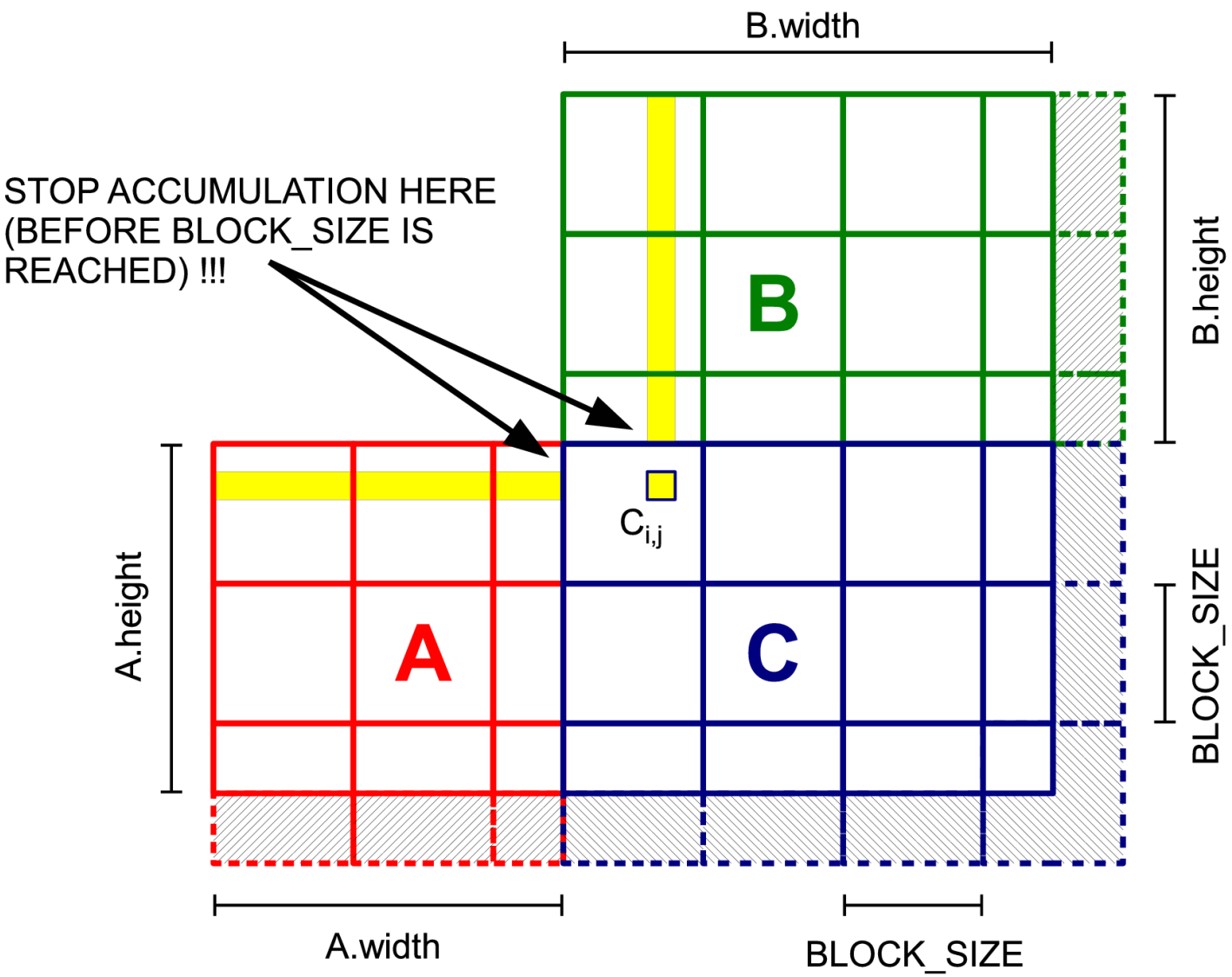
ALGORITHM

- Determine the number of submatrices along the width of **A** (or height of **B**) with size `BLOCK_SIZE*BLOCK_SIZE`
  → `mMax` (rounded up)
- Iterate over `m := 0…(mMax-1)`
  - ➢ Load submatrices of **A** and **B** with index `m` from global memory into local memory (every work-item is responsible for a single element of **A** and a single element of **B**)
  - ➢ Compute with every work-item a partial sum of $c_{i,j}$ from the submatrices stored in local memory (incl. handling of the border case if `A.width%BLOCK_SIZE != 0`)

Fig.: NVIDIA

Matrix Multiplication with Local Memory

Border Cases



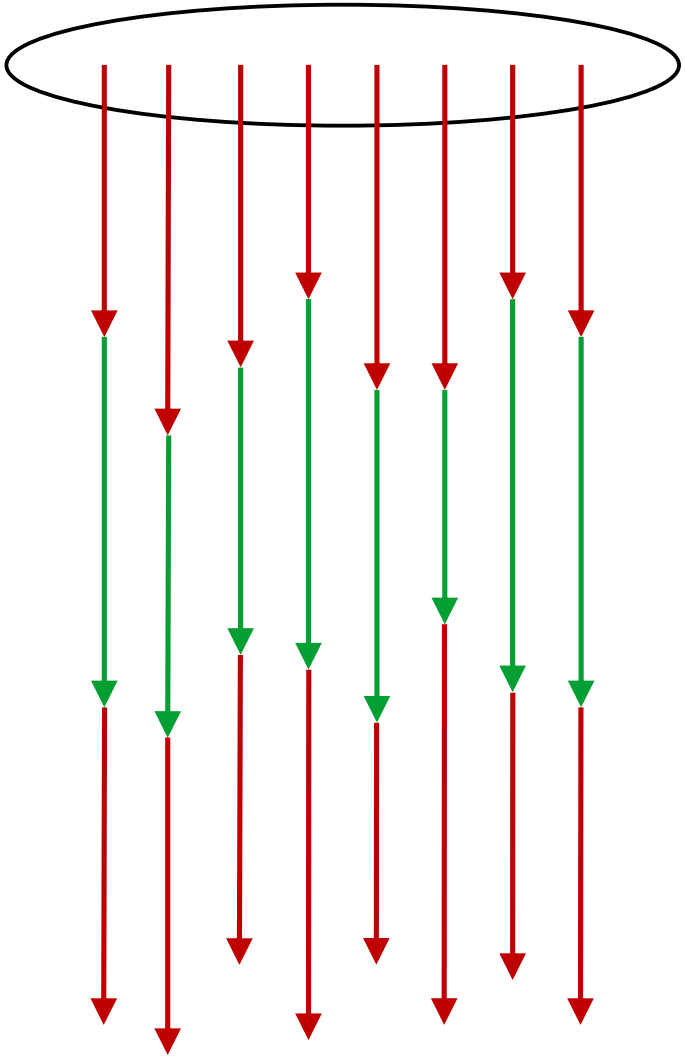
Accumulation over row/column pair:

Exercise 4: Tasks and Hints

- Think about “race conditions”; at which points shall all work-items in a work-group synchronize in the algorithm on slide 48?
- Add the corresponding synchronization calls in the already existing kernel source code!
- Do some benchmarking (kernel without local memory vs. kernel with local memory; try out several work-group sizes)!

- Host code to edit and modify (search for TODOs): `matmul_opencl.C`
- Device code to edit and modify (search for TODOs): `matmul_localMem.cl`
  - Kernel function: `matMulKernel_LM`
  - Synchronization function to use: `barrier(CLK_LOCAL_MEM_FENCE)`

Exercise 4: Hint on Race Condition
(Example for Nvidia GPU)



8 Nvidia warps
<u>within a work-group</u>
with 256 work-items

RED: Transfer from global into local memory (each work-item responsible for part of the data)

GREEN: Computations based on total content of local memory

RED: Transfer from global into local memory…

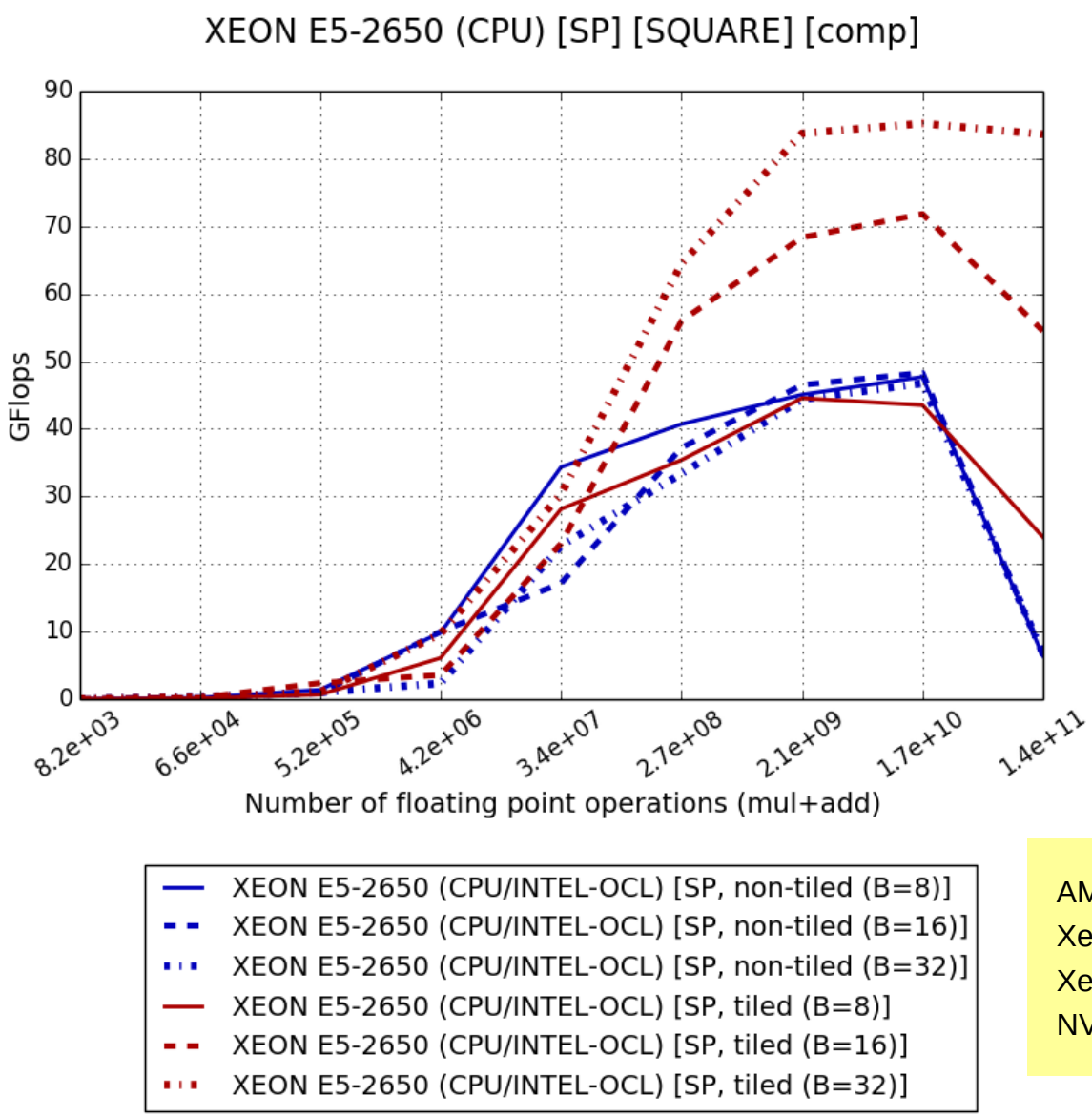
Would you expect correct computational results…?

Exercise 4: Solution

OpenCL kernel code

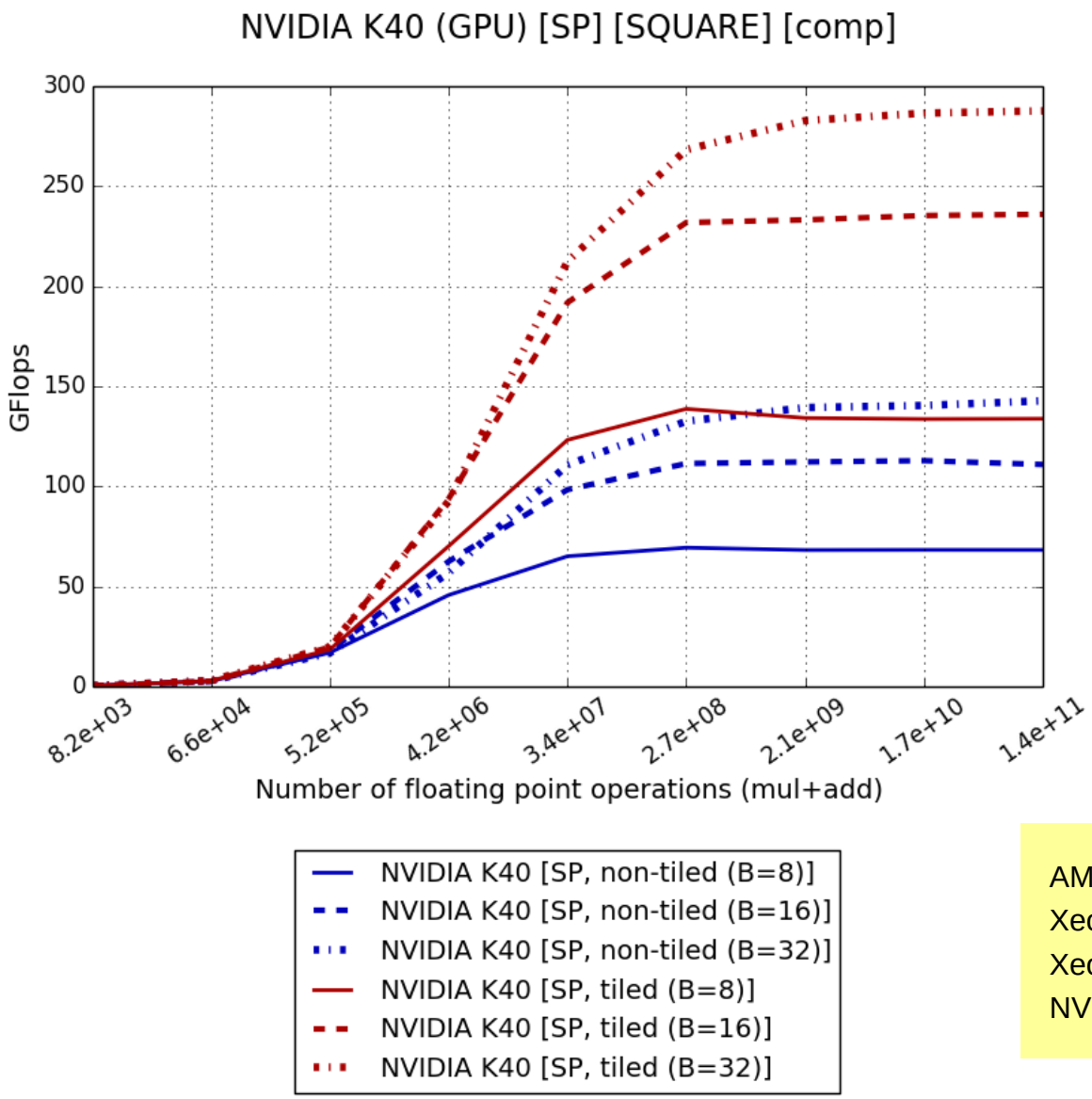
```
__kernel void matMulKernel_LM(...)
{
  ...

  for (int m = 0; m < mMax; ++m) {
    // Get base indices of sub-matrices Asub of A and Bsub of B
    Asub_baseIndex = Astride * BLOCK_SIZE * blockRow + BLOCK_SIZE * m;
    Bsub_baseIndex = Bstride * BLOCK_SIZE * m + BLOCK_SIZE * blockCol;

    // Load Asub and Bsub from global memory to local memory
    // Each thread loads one element of each sub-matrix
    Asub_index = Asub_baseIndex + row * Astride + col;
    if( Asub_index < A_numElems )
      As[row*BLOCK_SIZE+col] = Aelements[Asub_index];
    Bsub_index = Bsub_baseIndex + row * Bstride + col;
    if( Bsub_index < B_numElems )
      Bs[row*BLOCK_SIZE+col] = Belements[Bsub_index];

    // Synchronize to make sure the sub-matrices are loaded
    // before starting the computation
    barrier(CLK_LOCAL_MEM_FENCE);

    // Multiply row of Asub and column of Bsub together
    // (only iterate up to eMax to prevent inclusion of invalid elements from
    //  Asub and Bsub)
    int eMax;
    if( (m == (mMax-1)) && (r != 0) )
      eMax = r;
    else
      eMax = BLOCK_SIZE;
    for (int e = 0; e < eMax; ++e)
      Cvalue += As[row*BLOCK_SIZE+e] * Bs[e*BLOCK_SIZE+col];

    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    barrier(CLK_LOCAL_MEM_FENCE);
  }

  ...
}
```

Results on the CPU
(Exercise 4)



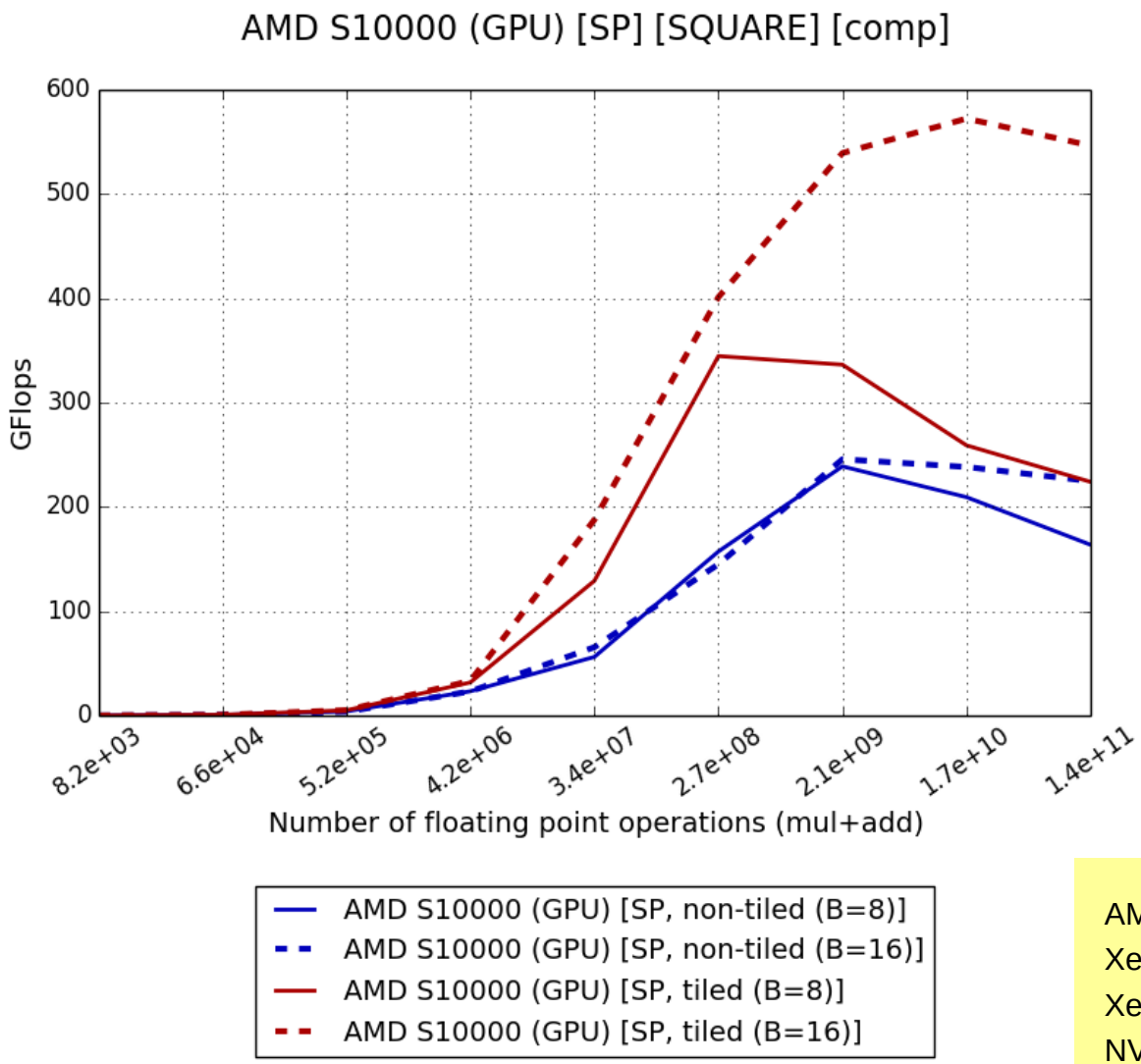
AMD S10000:	$R_{peak,SP} \approx 2900$ GFlops
Xeon Phi:	$R_{peak,SP} \approx 2000$ GFlops
Xeon E5-2650 (2x):	$R_{peak,SP} \approx 600$ GFlops
NVIDIA K40:	$R_{peak,SP} \approx 5000$ GFlops

Results on the Nvidia GPU
(Exercise 4)



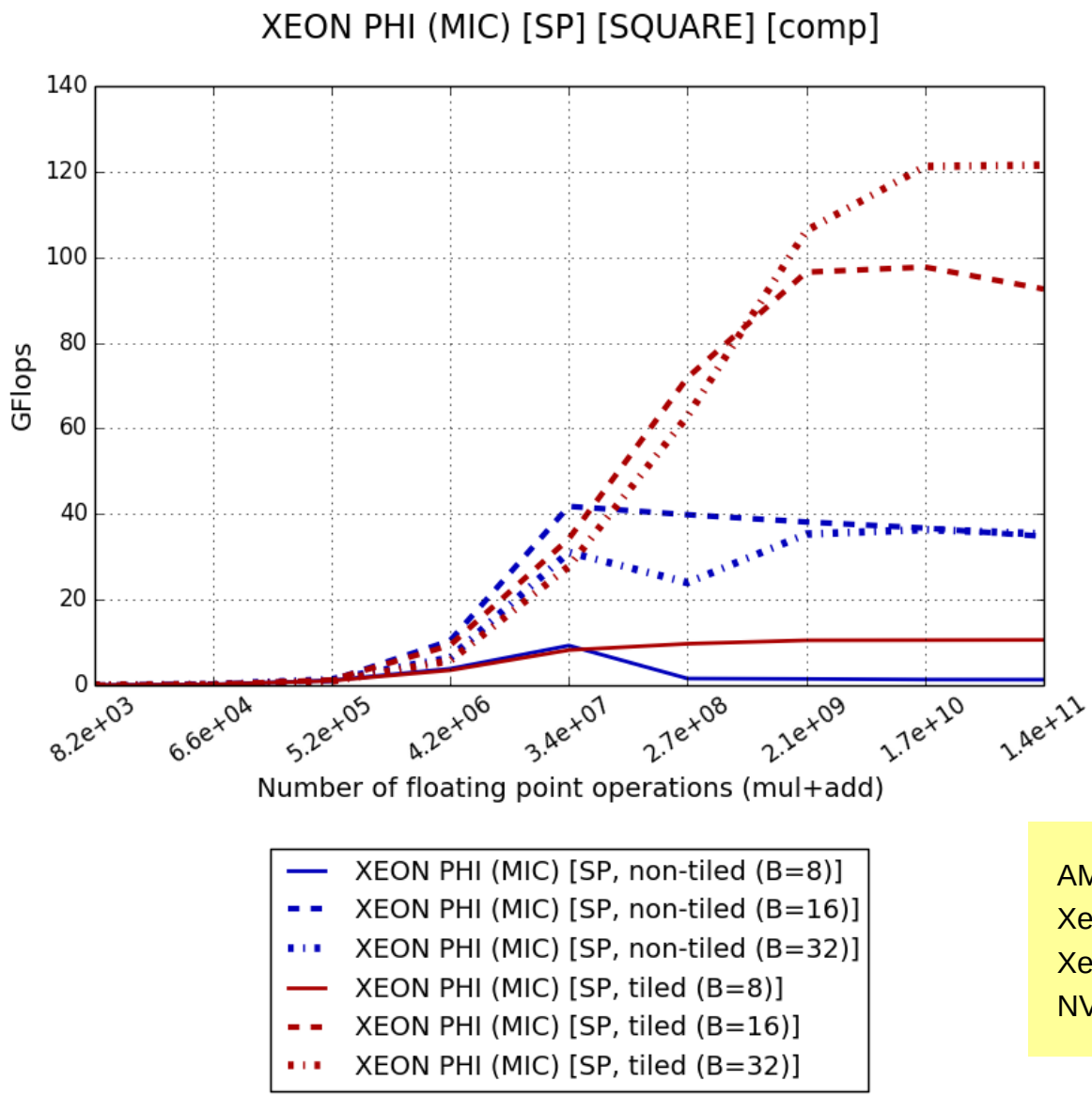
AMD S10000:	$R_{peak,SP} \approx 2900$ GFlops
Xeon Phi:	$R_{peak,SP} \approx 2000$ GFlops
Xeon E5-2650 (2x):	$R_{peak,SP} \approx 600$ GFlops
NVIDIA K40:	$R_{peak,SP} \approx 5000$ GFlops

Results on the AMD GPU
(Exercise 4)



AMD S10000:	$R_{peak,SP} \approx 2900$ GFlops
Xeon Phi:	$R_{peak,SP} \approx 2000$ GFlops
Xeon E5-2650 (2x):	$R_{peak,SP} \approx 600$ GFlops
NVIDIA K40:	$R_{peak,SP} \approx 5000$ GFlops

Results on Xeon Phi
(Exercise 4)



AMD S10000:	$R_{peak,SP} \approx 2900$ GFlops
Xeon Phi:	$R_{peak,SP} \approx 2000$ GFlops
Xeon E5-2650 (2x):	$R_{peak,SP} \approx 600$ GFlops
NVIDIA K40:	$R_{peak,SP} \approx 5000$ GFlops

Summary: Local Memory

- Significant improvement on all devices
  (even there where you would not expect it…)
  - Reason: Strongly reduced number of reads from slow global memory (applies in principle especially to GPUs)
- For best performance necessary to choose optimal work-group size
  - Optimal work-group size depends also on the problem size!

Further Tuning Techniques

Data Prefetching / Double Buffering

- …
- **Blue tile**: Register file → Local memory
- Synchronize work-items
- **Orange tile:** Global memory → Register file
- Compute **blue tile**
- Synchronize work-items
- Move on: **Orange tile** becomes **blue tile**, new **orange tile**
- …

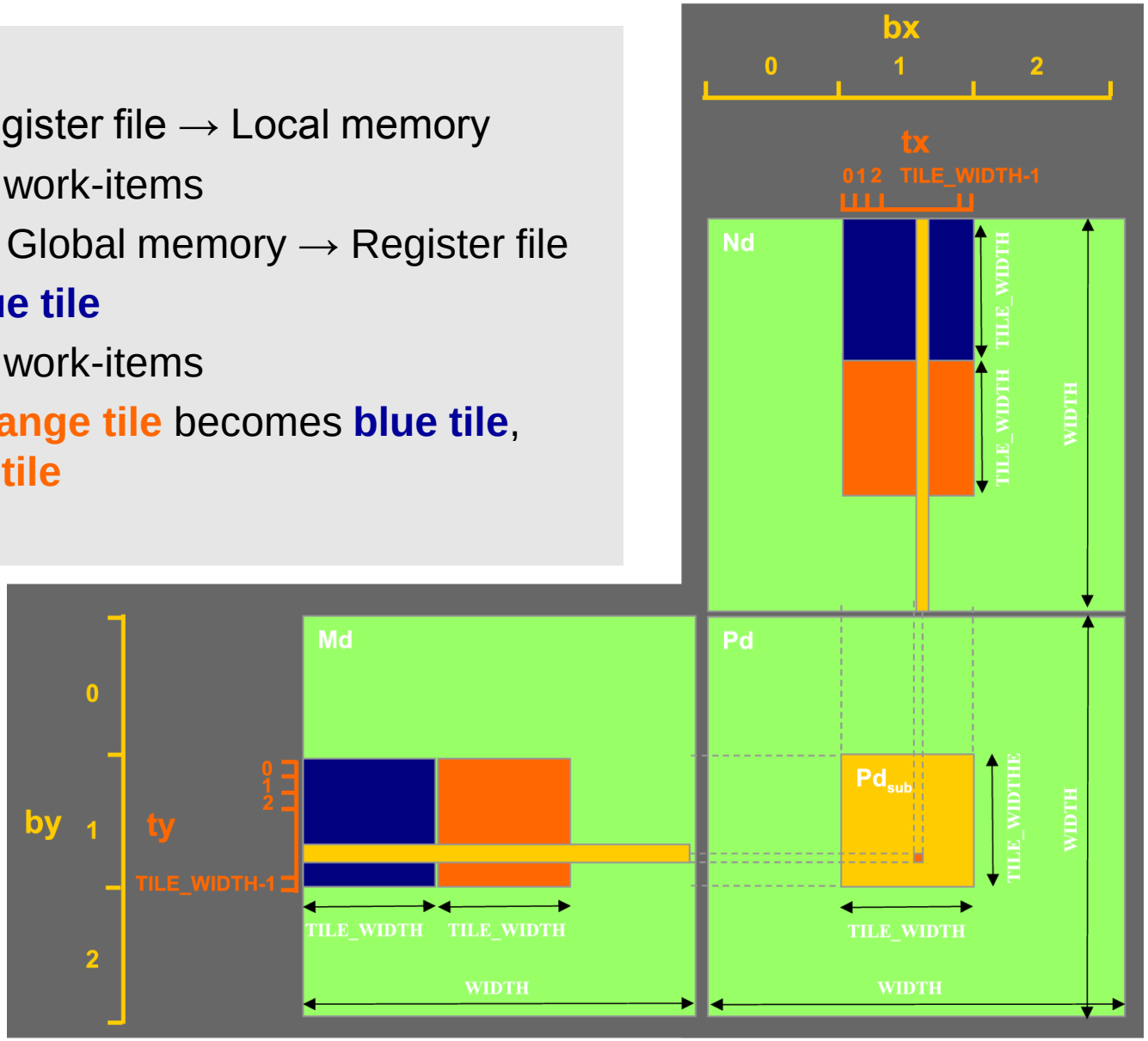


Fig.: Kirk & Hwu, ECE 498AL Lecture Slides

Decreasing Work-Item Granularity

- Every work-item computes two elements within C
  - Number of loads from global memory further reduced (from Matrix A [= Md])
  - Reduced non-computational overhead
  - ➔ More computational work per iteration done

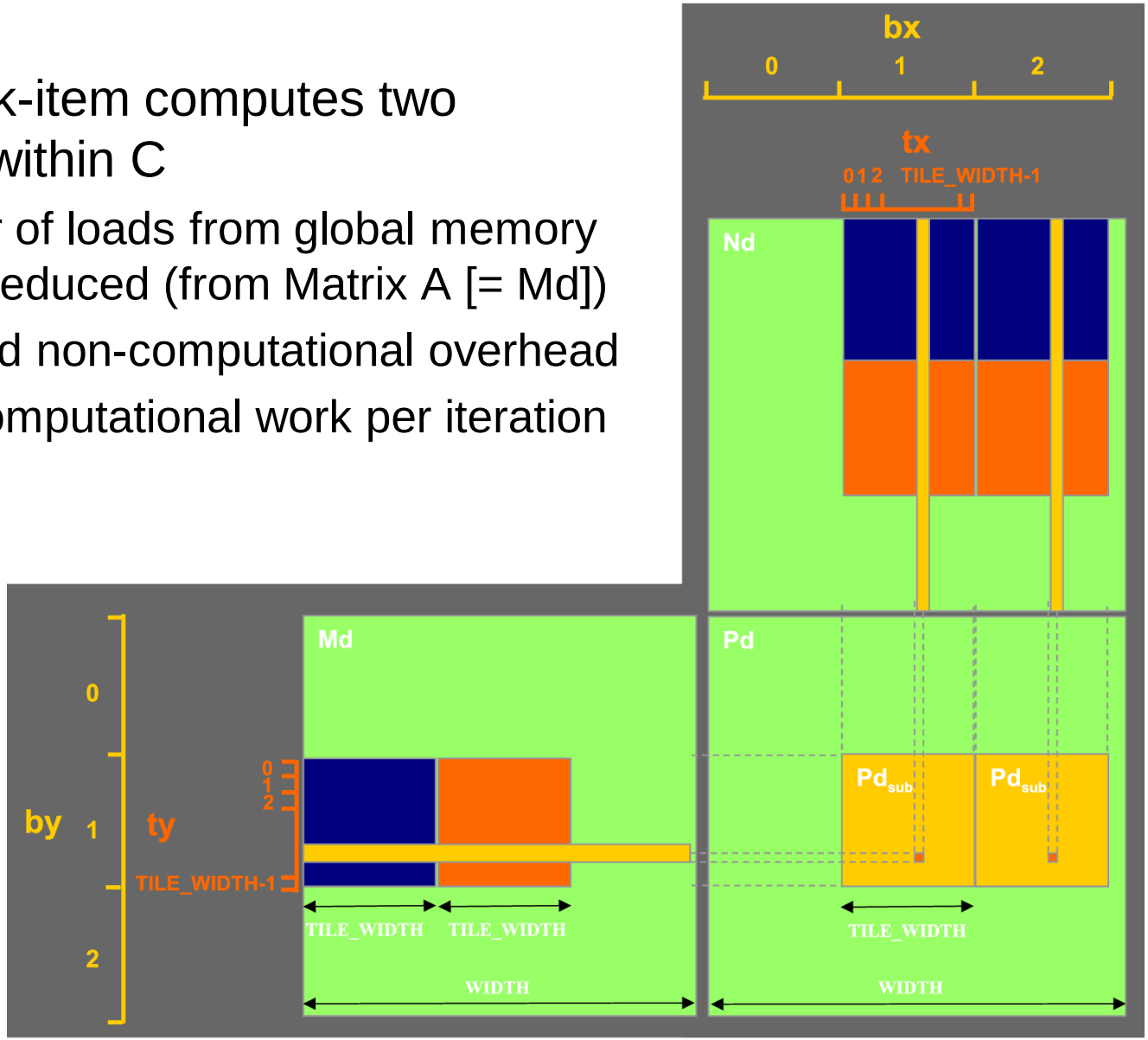
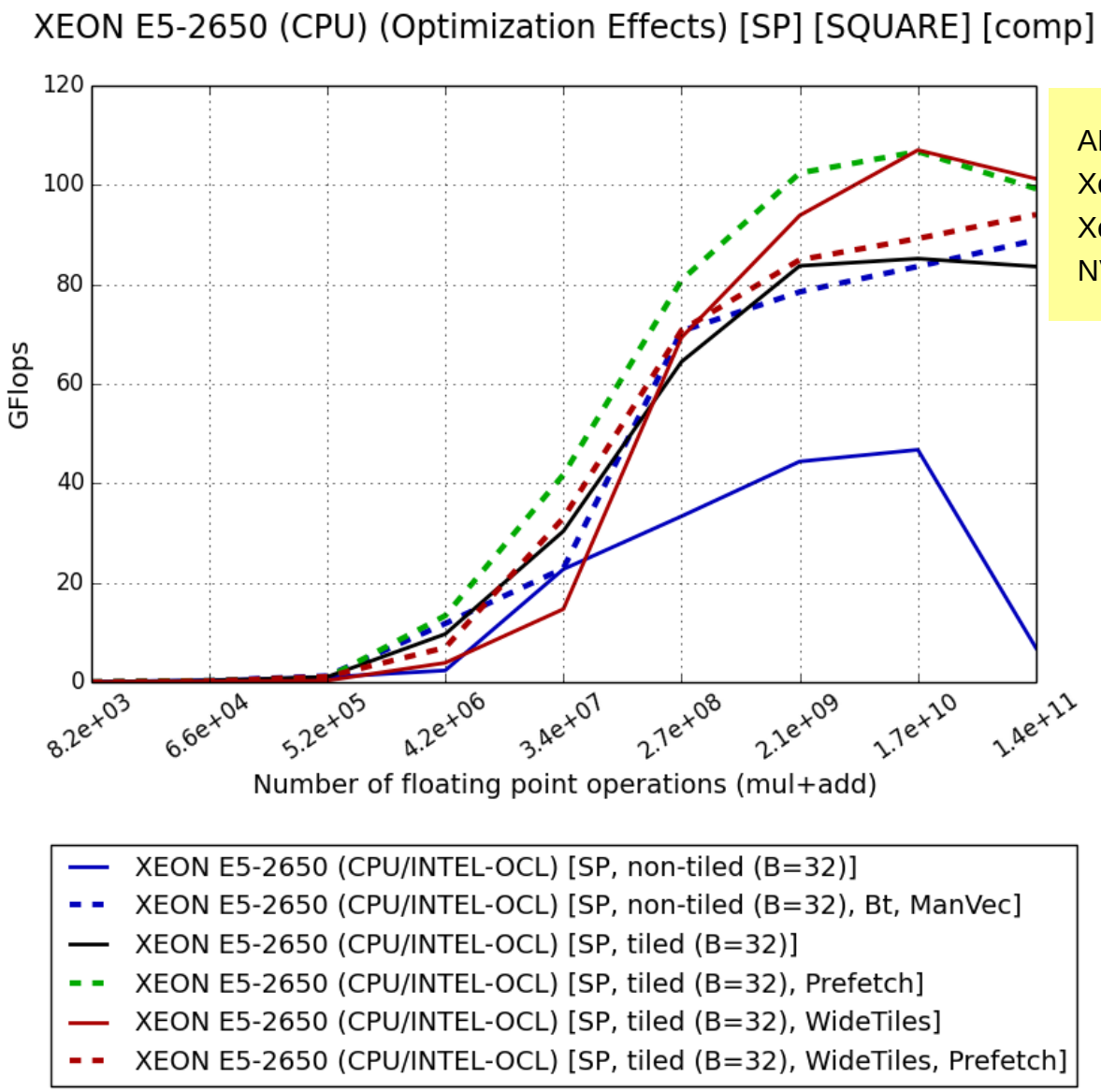


Fig.: Kirk & Hwu, ECE 498AL Lecture Slides

Results on the CPU

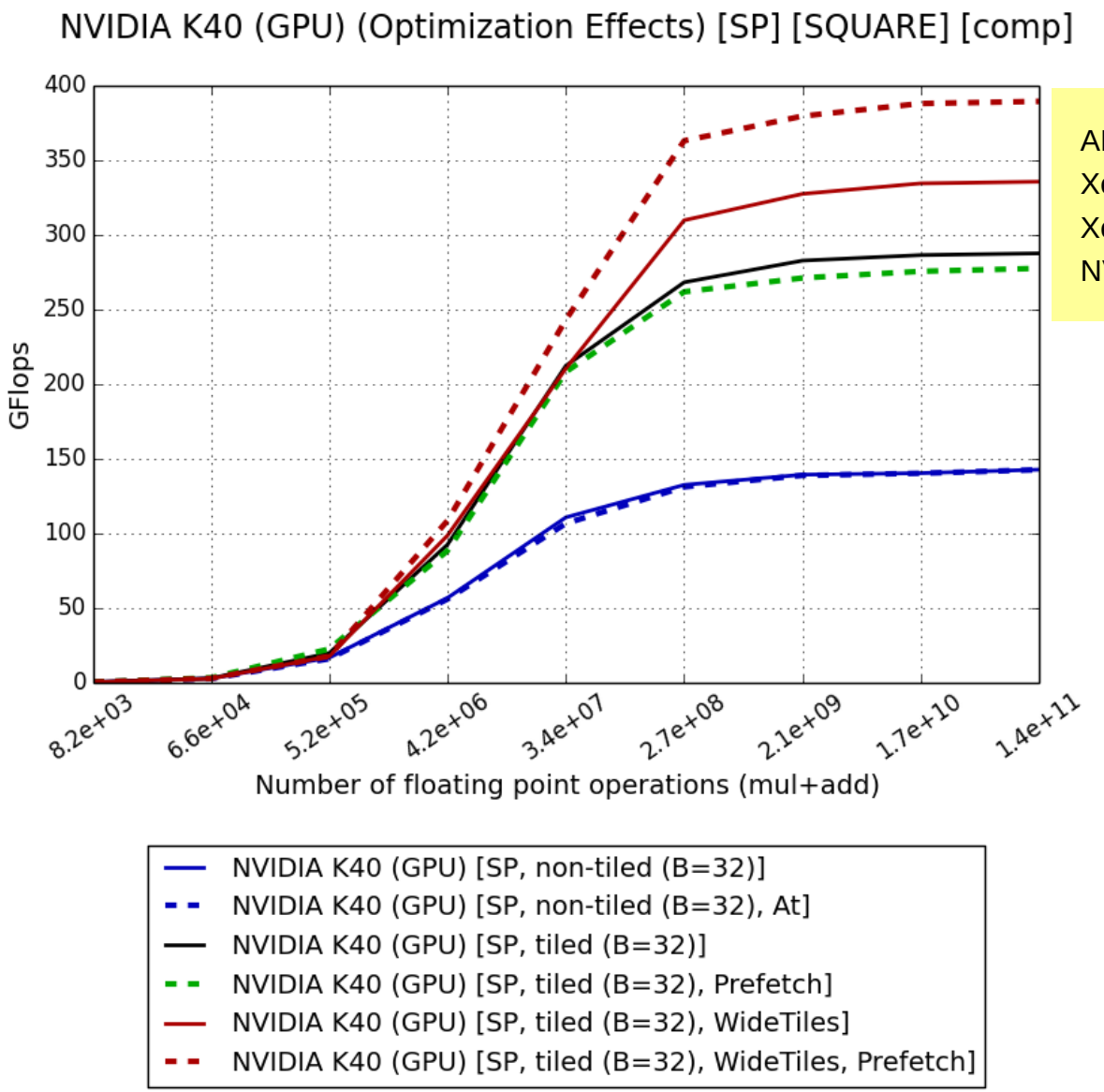
(Effects of Optimization for SP) [best work-group size selected]



AMD S10000:	$R_{peak,SP} \approx 2900$ GFlops
Xeon Phi:	$R_{peak,SP} \approx 2000$ GFlops
Xeon E5-2650 (2x):	$R_{peak,SP} \approx 600$ GFlops
NVIDIA K40:	$R_{peak,SP} \approx 5000$ GFlops

Results on the Nvidia GPU

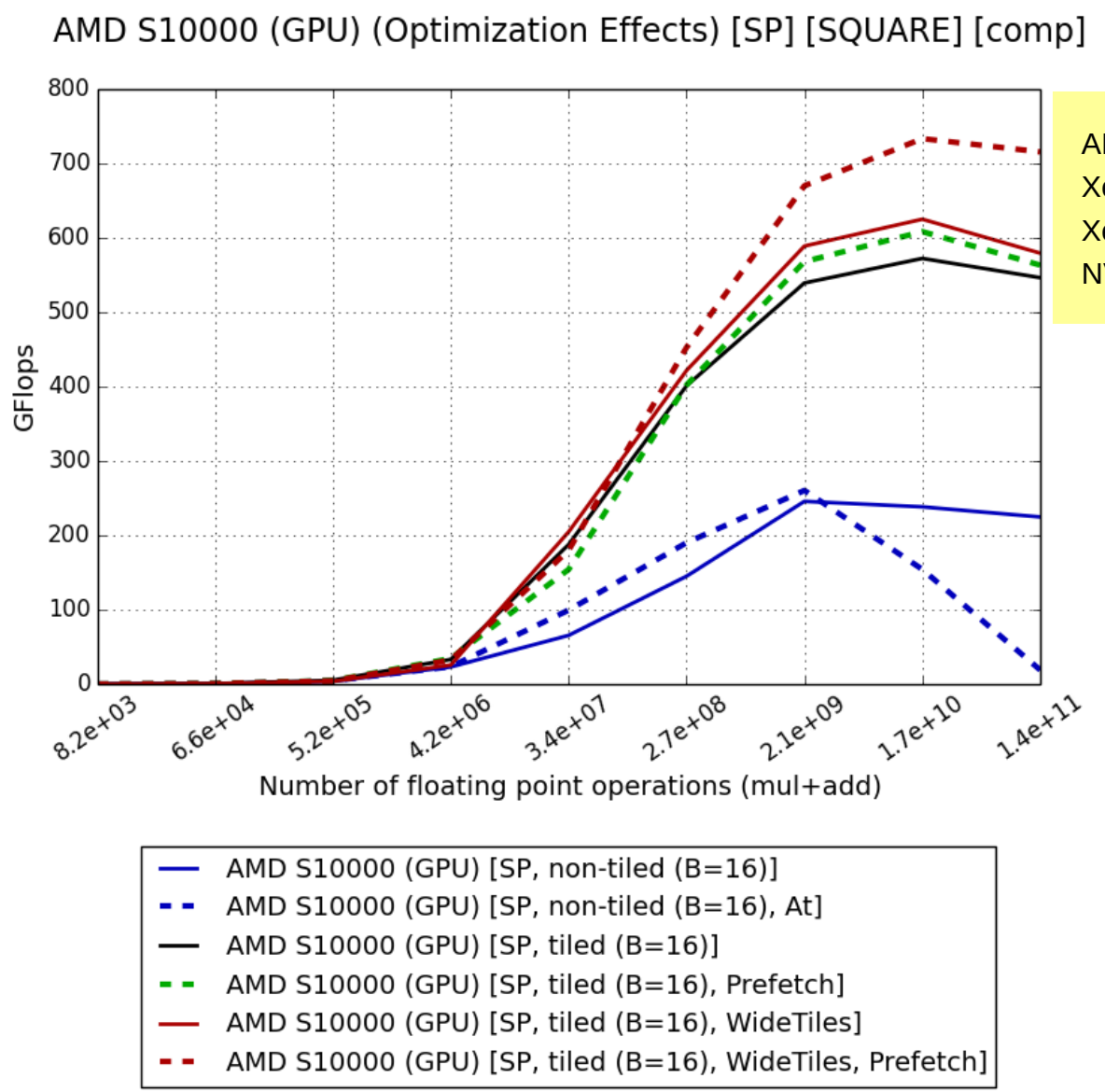
(Effects of Optimization for SP) [best work-group size selected]



AMD S10000:	$R_{peak,SP} \approx 2900$ GFlops
Xeon Phi:	$R_{peak,SP} \approx 2000$ GFlops
Xeon E5-2650 (2x):	$R_{peak,SP} \approx 600$ GFlops
NVIDIA K40:	$R_{peak,SP} \approx 5000$ GFlops

Results on the AMD GPU

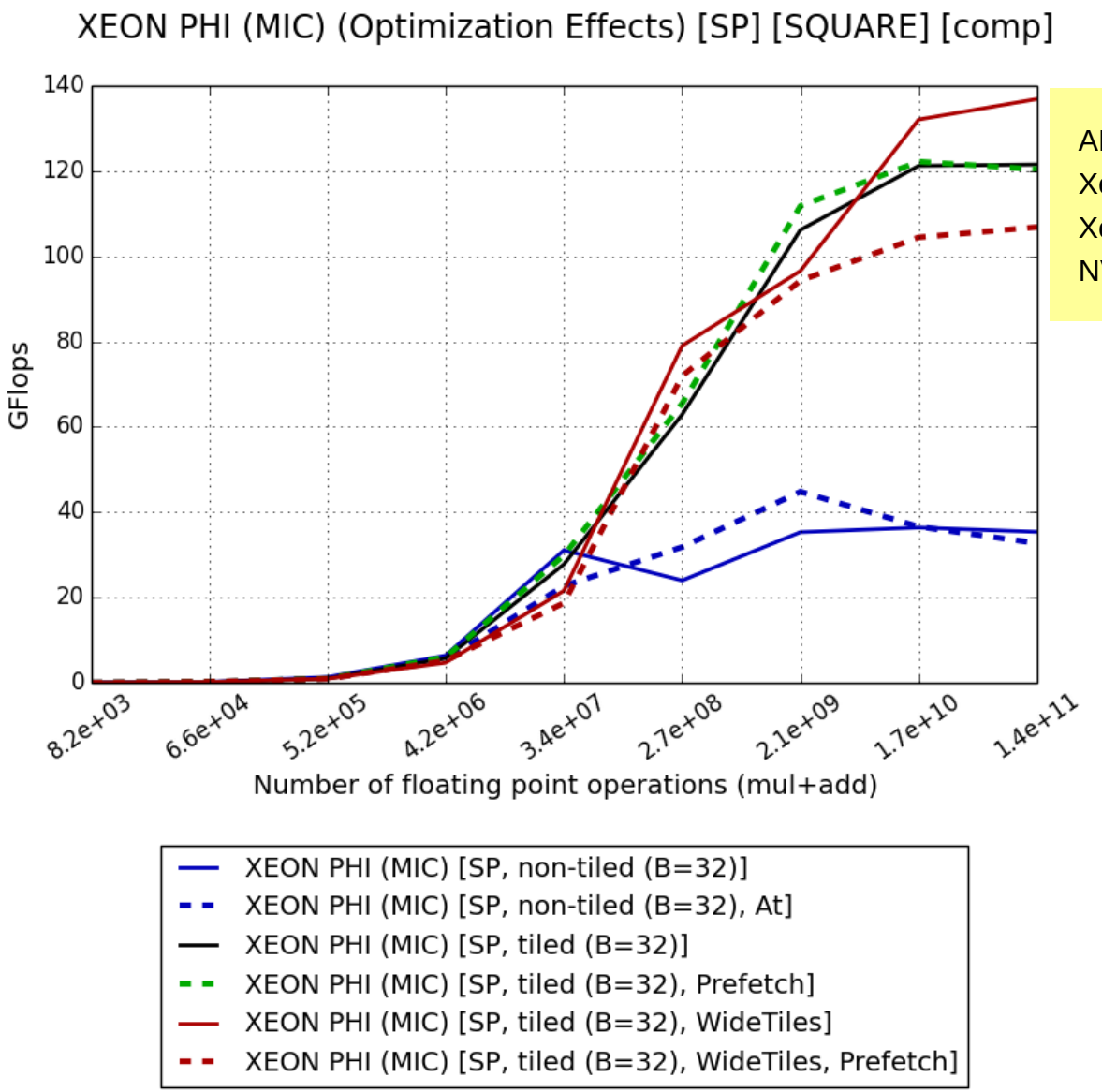
(Effects of Optimization for SP) [best work-group size selected]



AMD S10000:	$R_{peak,SP} \approx 2900$ GFlops
Xeon Phi:	$R_{peak,SP} \approx 2000$ GFlops
Xeon E5-2650 (2x):	$R_{peak,SP} \approx 600$ GFlops
NVIDIA K40:	$R_{peak,SP} \approx 5000$ GFlops

Results on Xeon Phi

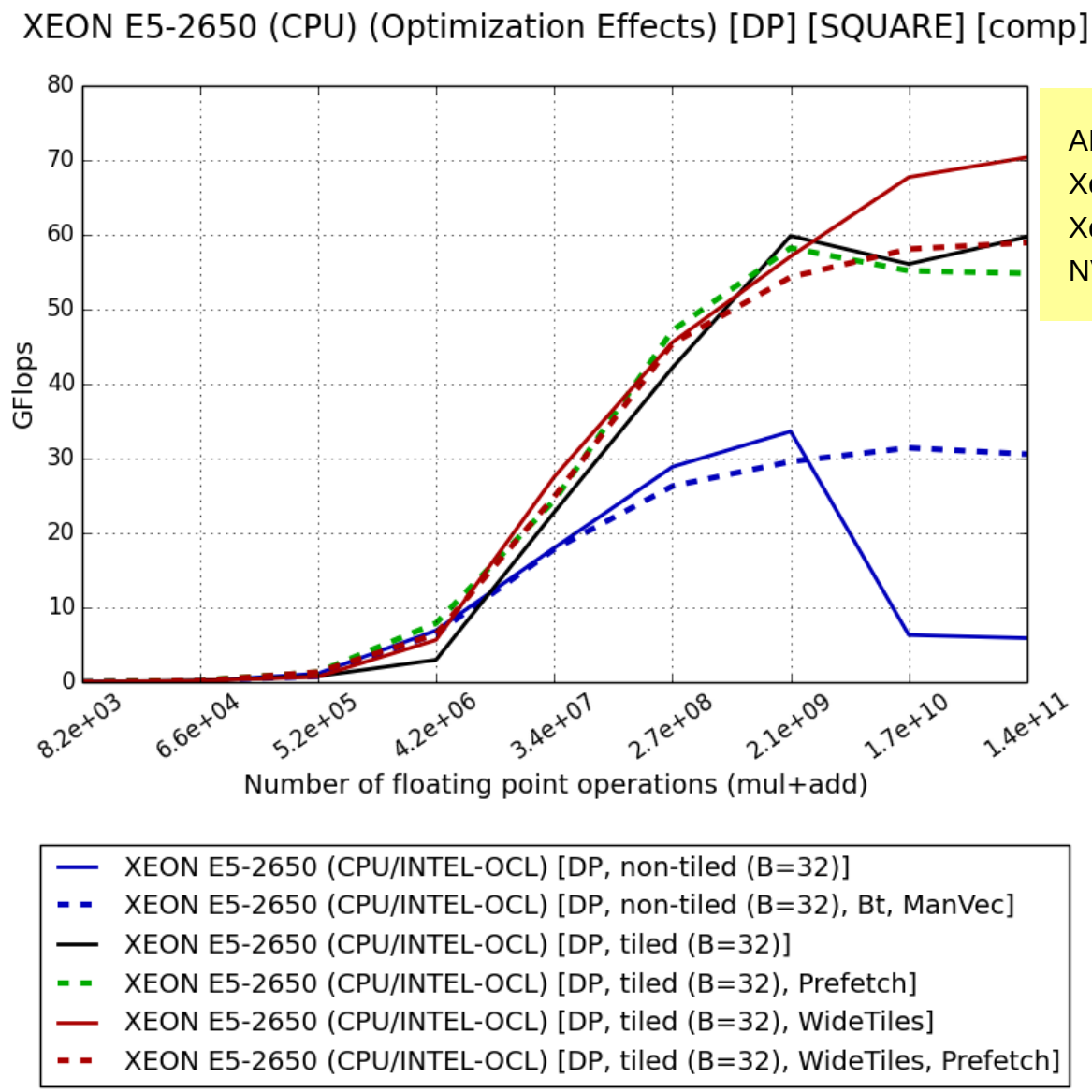
(Effects of Optimization for SP) [best work-group size selected]



AMD S10000:	$R_{peak,SP} \approx 2900$ GFlops
Xeon Phi:	$R_{peak,SP} \approx 2000$ GFlops
Xeon E5-2650 (2x):	$R_{peak,SP} \approx 600$ GFlops
NVIDIA K40:	$R_{peak,SP} \approx 5000$ GFlops

Results on the CPU

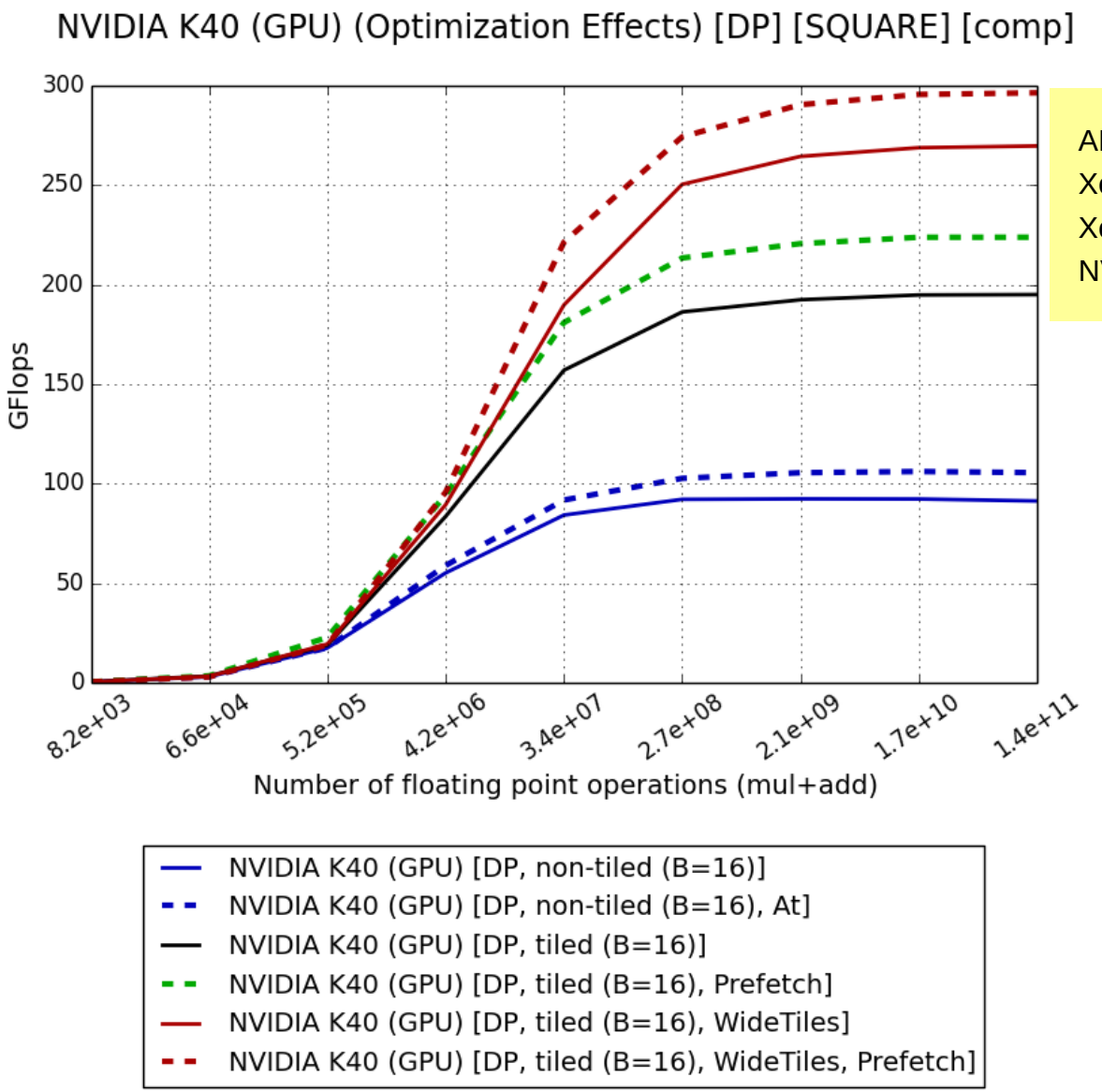
(Effects of Optimization for DP) [best work-group size selected]



AMD S10000:	$R_{peak,DP} \approx 730$ GFlops
Xeon Phi:	$R_{peak,DP} \approx 1000$ GFlops
Xeon E5-2650 (2x):	$R_{peak,DP} \approx 300$ GFlops
NVIDIA K40:	$R_{peak,DP} \approx 1660$ GFlops

Results on the Nvidia GPU

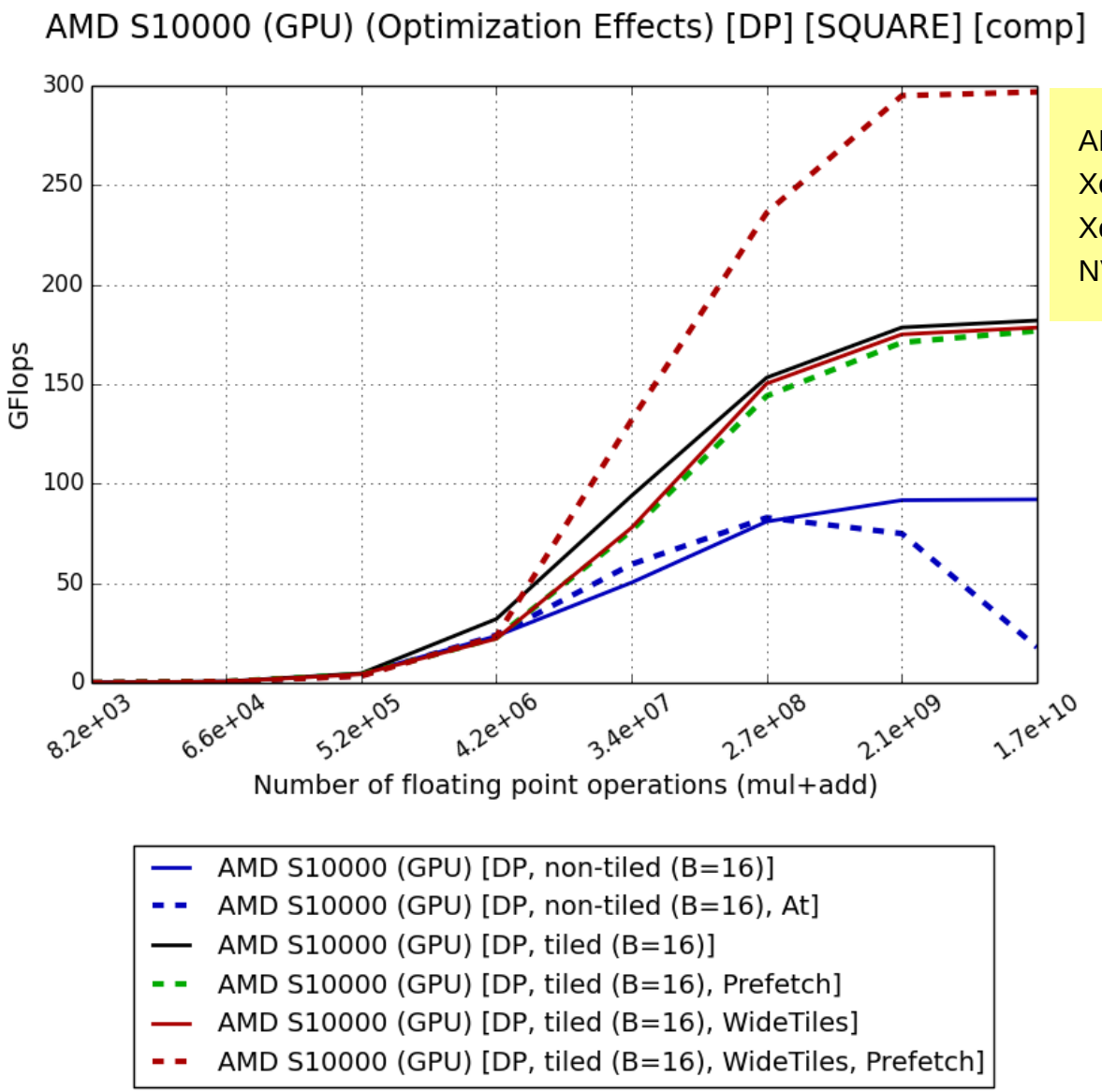
(Effects of Optimization for DP) [best work-group size selected]



AMD S10000:	$R_{peak,DP} \approx 730$ GFlops
Xeon Phi:	$R_{peak,DP} \approx 1000$ GFlops
Xeon E5-2650 (2x):	$R_{peak,DP} \approx 300$ GFlops
NVIDIA K40:	$R_{peak,DP} \approx 1660$ GFlops

Results on the AMD GPU

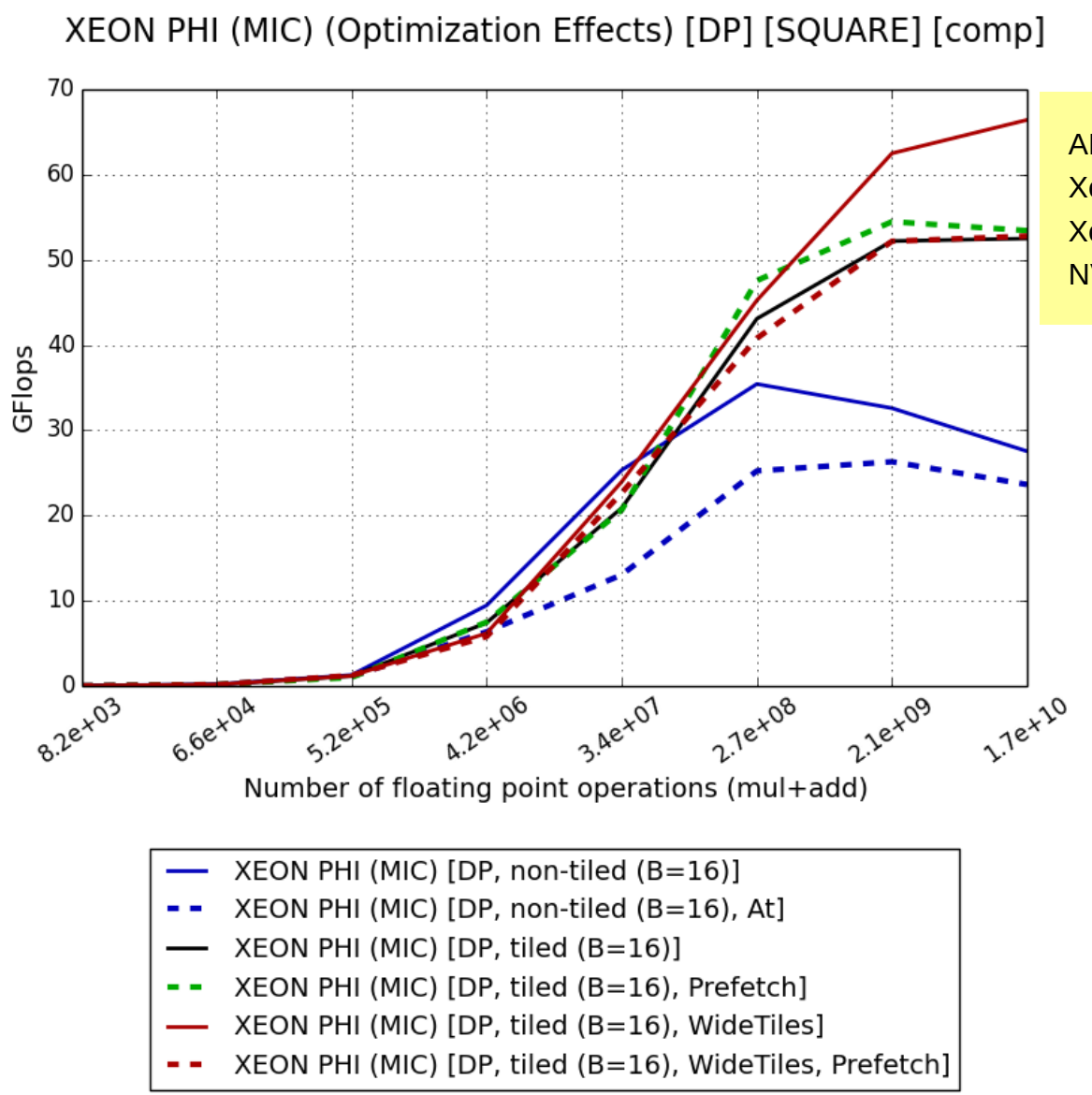
(Effects of Optimization for DP) [best work-group size selected]



AMD S10000:	$R_{peak,DP} \approx 730$ GFlops
Xeon Phi:	$R_{peak,DP} \approx 1000$ GFlops
Xeon E5-2650 (2x):	$R_{peak,DP} \approx 300$ GFlops
NVIDIA K40:	$R_{peak,DP} \approx 1660$ GFlops

Results on Xeon Phi

(Effects of Optimization for DP) [best work-group size selected]



AMD S10000:	$R_{peak,DP} \approx 730$ GFlops
Xeon Phi:	$R_{peak,DP} \approx 1000$ GFlops
Xeon E5-2650 (2x):	$R_{peak,DP} \approx 300$ GFlops
NVIDIA K40:	$R_{peak,DP} \approx 1660$ GFlops

Summary: Advanced Tuning

- Increasing computational workload per work-item usually improves performance
  - **BUT:** Always use a large enough number of work-items on GPUs to allow for latency hiding
- Prefetching often very useful technique on GPUs (less so on CPUs and MICs)
- Best results (not accounting for data transfer between device and host):
  - 25% of theoretical peak perf. on AMD GPU for SP
  - 40% of theoretical peak perf. on AMD GPU for DP

Overall Summary

Overall Summary

- **We have learned…**
  - …how the memory layout can influence performance in different ways on CPUs and GPUs
  - …how manual vectorization can give an additional performance boost to CPUs
  - …how local memory can be used to increase performance
  - …how to choose the optimal work-group size
  - …how to take care of synchronization between work-items
  - …how to apply more advanced tuning techniques
  - …*that it is never fully predictable from theory which mixture of tuning measures will give the best result on the device of your choice!*