GPU Hardware Performance II

Jiří Filipovič

Fall 2015

Recapitulation

Global memory

- warp should access data coalesced
- thread blocks should prevent partition camping

Shared memory

- threads in warp should access different banks (or the same data)

All memories

- sufficient occupancy needed to hide memory latencies

Matrix Transposition

From theoretical perspective:

- a trivial problem
- trivial parallelization
- trivially limited by the memory throughput (no arithmetic ops done)

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

Performance

When running the code on GeForce GTX 280 with large enough matrix 4000×4000, the throughput will be 5.3 GB/s

Where's the problem?

Performance

When running the code on GeForce GTX 280 with large enough matrix 4000×4000, the throughput will be 5.3 GB/s

Where's the problem?

Access to `odata` is interleaved. After modification (copy instead of transpose matrices):

```
odata[y*n + x] = idata[y*n + x];
```

the throughput is 112.4 GB/s. If `idata` is accessed in an interleaved way too, the resulting throughput would be 2.7 GB/s.

On Removing Interleaving

The matrix can be processed per tiles

- we read the tile into the shared memory row-wise
- we will store its transposition into the global memory row-wise
- thus having both reading and writing without interleaving

On Removing Interleaving

The matrix can be processed per tiles

- we read the tile into the shared memory row-wise
- we will store its transposition into the global memory row-wise
- thus having both reading and writing without interleaving

What size of tiles should be used?

- lets consider square tiles for simplicity
- for aligned reading, the row size has to be multiple of 16
- we can consider tile sizes of 16×16, 32×32, and 48×48 because of shared memory size limitations
- best size can be determined experimentally

Tiled Transposition

```
__global__ void mtran_coalesced(float *odata, float *idata, int n)
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = x + y*n;
  x = blockIdx.y * TILE_DIM + threadIdx.x;
  y = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = x + y*n;

  for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS)
    tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*n];

  __syncthreads();

  for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS)
    odata[index_out+i*n] = tile[threadIdx.x][threadIdx.y+i];
}
```

Performance

The highest performance was measured for 32×32 tile size and 32×8 thread block size – 75.1 GB/s

Performance

The highest performance was measured for 32×32 tile size and 32×8 thread block size – 75.1 GB/s

- that's significantly better but still far from simple copying

Performance

The highest performance was measured for 32×32 tile size and 32×8 thread block size – 75.1 GB/s

- that's significantly better but still far from simple copying
- the kernel is more complex, contains synchronization
  - we need to figure out whether we got the maximum or there's still a problem somewhere

Performance

The highest performance was measured for 32×32 tile size and 32×8 thread block size – 75.1 GB/s

- that's significantly better but still far from simple copying
- the kernel is more complex, contains synchronization
  - we need to figure out whether we got the maximum or there's still a problem somewhere
- if we only copy within the blocks, we get 94.9GB/s
  - something is still sub-optimal

Shared Memory

When reading from the global memory, we write into the shared memory row-wise

```
tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*n];
```

Shared Memory

When reading from the global memory, we write into the shared memory row-wise

```
tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*n];
```

When writing to the global memory, we read from the shared memory column-wise

```
odata[index_out+i*n] = tile[threadIdx.x][threadIdx.y+i];
```

That's reading with interleaving which is multiple of 16, the whole column is in a single memory bank – thus creaing 16-way bank conflict.

Shared Memory

When reading from the global memory, we write into the shared memory row-wise

```
tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*n];
```

When writing to the global memory, we read from the shared memory column-wise

```
odata[index_out+i*n] = tile[threadIdx.x][threadIdx.y+i];
```

That's reading with interleaving which is multiple of 16, the whole column is in a single memory bank – thus creaing 16-way bank conflict.
A solution is padding:

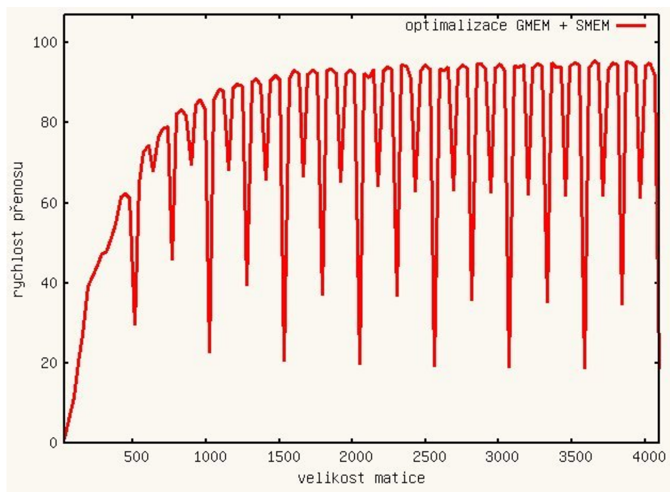
```
__shared__ float tile[TILE_DIM][TILE_DIM + 1];
```

Performance

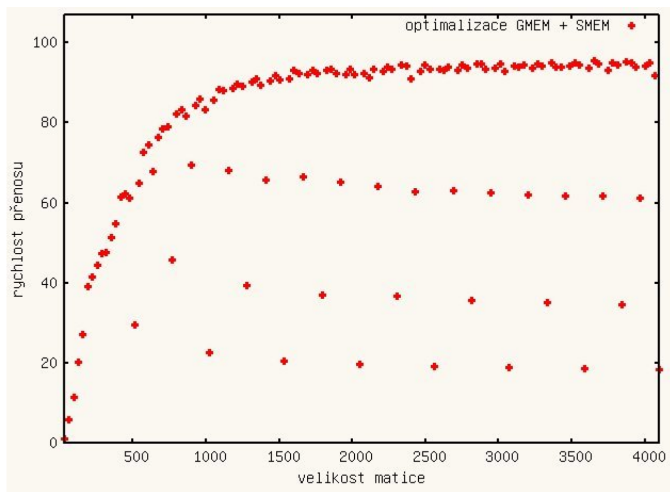
Now our implementations shows 93.4 GB/s.

- as good as simple copying
- it seems we can't do much better for given matrix
- beware of different input data sizes (partition camping)

Performance



Performance



Performance Drops

The performance drops for some size and the behavior is regular

Performance Drops

The performance drops for some size and the behavior is regular

- for matrices sized multiple of 512, we only get 19 GB/s
- for other matrices sized multiple of 256, we only get 35 GB/s
- for other matrices sized multiple of 128, we only get 62 GB/s

Performance Drops

One memory region has width of 2 tiles (256 B / 4 B per float, 32 floats in a tile). If we analyze tiles placement w.r.t. matrix size, we learn that

- with multiple of 512 size, the tiles in the same column are in the same region
- with multiple of 256 size, each column is at most in two regions
- with multiple of 128, each column is at most in four regions

We have discovered partition camping.

How to Remove Partition Camping?

We can pad matrices and avoid bad matrix sizes.

- more complicated work with such implementation (all kernels accessing matrix have to implement padding, or we need to convert matrix)
- it occupies more memory

How to Remove Partition Camping?

We can pad matrices and avoid bad matrix sizes.

- more complicated work with such implementation (all kernels accessing matrix have to implement padding, or we need to convert matrix)
- it occupies more memory

We can change the mapping of thread blocks id's on matrix tiles

- diagonal mapping ensures access to different regions

```
int blockIdx_y = blockIdx.x;
int blockIdx_x = (blockIdx.x+blockIdx.y) % gridDim.x;
```

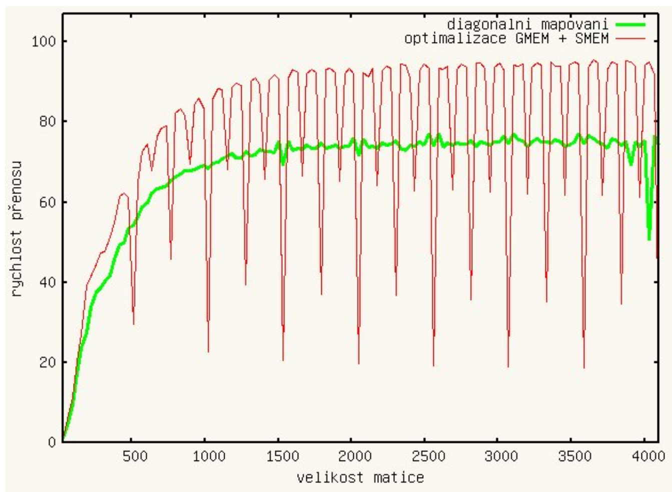
Performance

New implementation gives 80 GB/s

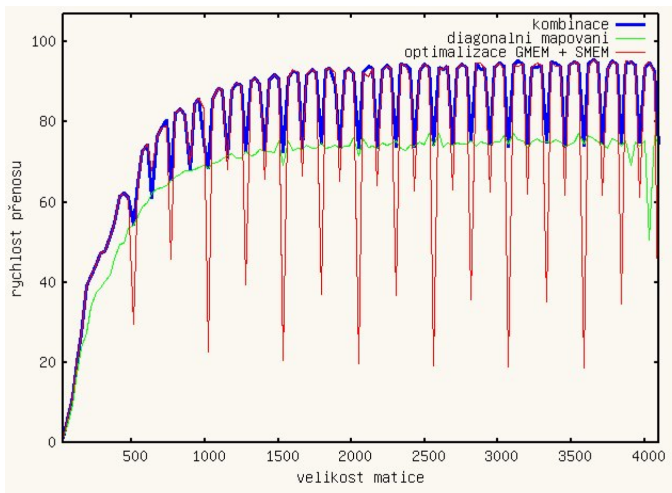
- performance doesn't drop where we saw it previously
- for matrix size of multiple of 128 still worse then the original implementation
  - the algorithm is more complex
- we can use it only for the problematic data sizes

For given problem, there may not be (and often there is not) an ideal algorithm for the whole input data size range. It is necessary to benchmark as not all the problems are easily revealed just by looking at the code.

Performance



Performance



Performance Summary

All optimizations were only toward better accommodation of HW properties

- still we got $17.6\times$ speedup
- when creating an algorithm, it is necessary to understand HW limitations
- otherwise we wouldn't have to develop specifically for GPUs – developing a good sequential algorithm would have been just fine. . .

Optimizations Effects

Beware of optimization effects

- if we took 4096×4096 matrices instead of 4000×4000, the memory bank conflict removal would have been just marginal
- after removing partition camping, the effect of memory bank conflicts becomes visible
- thus it makes sense to go from more general/substantial optimizations to the less general ones
- if some (provably correct) optimization does not result in performance increase, we need to analyze, what limits the algorithm performance

Processing of Instructions

Processing of instructions on a multiprocessor (c. c. 1.x)

- there are 8 SP cores and 2 SFU cores
- if the SP and SPU instruction processing is not overlapped, the multiprocessor can process up to 8 instructions per cycle
  - one warp is thus done in 4 or more cycles
- some instructions are significantly slower
- knowledge of instruction processing time helps us to design efficient code

Floating Point Operations

GPU is designed as graphical HW

- graphical operations mostly use floating point numbers
- efficiently implemented in GPUs
- newer GPUs (c. c. ≥ 1.3) can work in double precision while older ones in single precision only
- some arithmetic operations are used very frequently in graphics
    - GPU implements them in HW
    - HW implementation provides lower precision (not in issue for lots of applications)
    - differentiated using "__" prefix

Arithmetic Operations

Floating point operations

- addition, multiplication very fast
- multiplication and addition may be combined into a single MAD instruction for c. c. 1.x
  - lower precision
  - 1 cycle speed on SP
  - *__fadd_rn()* and *__fmul_rn()* may be used to enforce avoiding MAD instruction during compilation
- MAD is replaced by FMAD for c. c. $\geq$ 2.0 (same speed, higher precision)
- 64b versions at lower speed: 1/8 (1.3), 1/2 (2.0), 1/12 (2.1), 1/24 (3.0), 1/3 (3.5), 1/32 (5.x)
- division is relatively slow, reciprocal is faster

Arithmetic Operations

Transcendental functions

- *__sinf(x)*, *__cosf(x)*, *__expf(x)*
- *sinf(x)*, *cosf(x)*, *expf(x)* more precise but an order of magnitude slower
- other operations with different speed and precision trade-offs are implemented, see CUDA manual

Integer operations

- addition as for the floating point ops
- multiplication on c. c. 1.x 2 instructions on an MP
  - *__mul24(x, y)* a *__umul24(x, y)* 8 instructions
- multiplication on c. c. 2.x is as fast as floating point ops, 24-bit version is slow
- division and modulo is very slow, but if n is power of 2, we can utilize
  - i/n is equivalent to $i >> log2(n)$
  - $i\%n$ is equivalent to $i\&(n-1)$

Loops

Small loops have significant overhead

- jumps
- conditions
- control variable updates
- significant part of instructions may be pointer arithmetics
- low ILP

Loop unrolling is an option

- partially may be done by the compiler
- we can do manual unrolling or use *#pragma unroll*

Other Instructions

Other common instructions are performed at the basic speed (i.e., correspond to number of SPs)

- comparison
- bit operations
- memory access instructions (given the limitations discussed earlier and memory latency/bandwidth)
  - the offset may be register value + constant for 32-bit addressing (higher overhead for 64-bit addressing)
- synchronization (unless we get blocked)

Beware of Shared Memory

If memory bank conflict is avoided, the shared memory is as fast as registers at c.c. 1.x
But beware

- instructions can work with only one operand in the shared memory
- if more than one operands in shared memory are used for one instruction, explicit load/store is necessary
- MAD instructions run slower (c.c. 1.x)
  - $a + s[i]$ 4 cycles per warp
  - $a + a * s[i]$ 5 cycles per warp
  - $a + b * s[i]$ 6 cycles per warp
- these details are not published by NVIDIA (revealed through measurements)
- interesting only for really critical code

Beware of Shared Memory

Newer GPUs have relatively slower shared memory (comparing to register speed)

- Fermi and Maxwell have lower bandwidth even if one operand in shared memory is accessed
- Kepler uses only 1/2 of available bandwidth for 32-bit access

C for CUDA Compilation

Device code can be compiled into PTX assembler and binary files

- PTX is intermediate code, does not correspond directly to GPU instructions
  - easier to read
  - harder to figure out what really happens on GPU

Binary files may be disassembled using *cuobjdump* tool

- for GT200 and newer
- *decuda* for older GPUs (may not work completely reliably)

Naive Implementation

```
__global__ void mmul(float *A, float *B, float *C, int n){
  int x = blockIdx.x*blockDim.x + threadIdx.x;
  int y = blockIdx.y*blockDim.y + threadIdx.y;

  float tmp = 0;
  for (int k = 0; k < n; k++)
    tmp += A[y*n+k] * B[k*n+x];

  C[y*n + x] = tmp;
}
```

Recapitulation

Naive implementation

- each thread computes one element of the resulting matrix
- memory-bound
- theoretical peak 66.8 GFlops
- performance depends on threads arrangement – blocks 128×1: 36.6 GFlops, blocks 1×128: 3.9 GFlops

Recapitulation

Naive implementation

- each thread computes one element of the resulting matrix
- memory-bound
- theoretical peak 66.8 GFlops
- performance depends on threads arrangement – blocks 128×1: 36.6 GFlops, blocks 1×128: 3.9 GFlops

Now, we understand the results

- theoretical maximum cannot be reached – we access GPU memory in at least 32-byte chunks, so reading from A is not efficient
- blocks 128×1 result in coalesced access into B, blocks 1×128 result in strided access

Recapitulation

We have implemented a tiled algorithm

- each thread block read tiles from A, B into shared memory, exploit data locality (data are moved into shared memory once and read many times)
- theoretical peak 568 GFlops, we have reached 198 GFlops

We can improve the implementation...

Tiled Algorithm

```
__global__ void mmul(float *A, float *B, float *C, int n){
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  __shared__ float As[BLOCK][BLOCK];
  __shared__ float Bs[BLOCK][BLOCK];

  float Csub = 0.0f;
  for (int b = 0; b < n/BLOCK; b++){
    As[ty][tx] = A[(ty + by*BLOCK)*n + b*BLOCK+tx];
    Bs[ty][tx] = B[(ty + b*BLOCK)*n + bx*BLOCK+tx];
    __syncthreads();

    for (int k = 0; k < BLOCK; k++)
      Csub += As[ty][k]*Bs[k][tx];
    __syncthreads();
  }

  C[(ty + by*BLOCK)*n + bx*BLOCK+tx] = Csub;
}
```

Implementation Pitfalls

```
As[ty][tx] = A[(ty + by*BLOCK)*n + b*BLOCK+tx];
Bs[ty][tx] = B[(ty + b*BLOCK)*n + bx*BLOCK+tx];
...
C[(ty + by*BLOCK)*n + bx*BLOCK+tx] = Csub;
```

Global memory access is OK.

```
Csub += As[ty][k]*Bs[k][tx];
```

Also shared memory access is OK.

- if a thread block *x*-size is multiple of warp size, variable *As* is broadcasted
- array *Bs* is read in contiguous lines, which is conflict-free

Theoretical Peak

Can we be more precise in theoretical peak computation?

- we have used a theoretical peak of GPU in MAD instructions (622 GFlops)
- now, we know that MAD instructions with operand in shared memory are 50% slower
- the more precise theoretical bound is 415 GFlops
- our implementation is still far from that

Performance Pitfalls

What causes performance degradation?

- overhead of kernel execution, thread creation
  - mainly for fast kernels and low instructions per thread
  - threads can do more work in serial
- instruction overhead
  - pointer arithmetics, loops
  - can be reduced
- synchronization
  - may or may not be an issue
- load/store in computation
  - two operands in SMEM per one MAD instruction

If we count the performance bound for one load per MAD with operand in SMEM, we get 244 GFlops.

Searching for Better Implementation

Can be a number of load instructions decreased?

Searching for Better Implementation

Can be a number of load instructions decreased?

- exploiting data locality in shared memory decreases global memory pressure

Searching for Better Implementation

Can be a number of load instructions decreased?

- exploiting data locality in shared memory decreases global memory pressure
- exploiting data locality in registers decreases shared memory pressure

Searching for Better Implementation

Can be a number of load instructions decreased?

- exploiting data locality in shared memory decreases global memory pressure
- exploiting data locality in registers decreases shared memory pressure
- how to do it? we reduce number of threads and assign more work to them

Searching for Better Implementation

Can be a number of load instructions decreased?

- exploiting data locality in shared memory decreases global memory pressure
- exploiting data locality in registers decreases shared memory pressure
- how to do it? we reduce number of threads and assign more work to them

Thread block of size $m \times n$ will process tile of size $m \times m$, where $m = n \cdot k; k \in N$.

- large m potentially increases synchronization overhead
- small m reduces shared memory locality
- small n reduces available parallelism
- we will find value for m and n experimentally

Searching for Better Implementation

Best results found for $m = 32$, $n = 16$ (32×16 blocks working with 32×32 tiles).

- one load to two MAD instructions results in theoretical bound 311 GFlops
- we have 235.4 GFlops
- something is wrong

Code disassembly

We focus on the inner loop

```
Csub1 += As[ty][k]*Bs[k][tx];
Csub2 += As[ty+16][k]*Bs[k][tx];
```

```
...
mov.b32 $r0, s[$ofs4+0x0000]
add.b32 $ofs4, $ofs2, 0x00000180
mad.rn.f32 $r7, s[$ofs1+0x0008], $r0, $r7
mad.rn.f32 $r8, s[$ofs3+0x0008], $r0, $r8
...
```

Code disassembly

We focus on the inner loop

```
Csub1 += As[ty][k]*Bs[k][tx];
Csub2 += As[ty+16][k]*Bs[k][tx];
```

```
...
mov.b32 $r0, s[$ofs4+0x0000]
add.b32 $ofs4, $ofs2, 0x00000180
mad.rn.f32 $r7, s[$ofs1+0x0008], $r0, $r7
mad.rn.f32 $r8, s[$ofs3+0x0008], $r0, $r8
...
```

Compiler was able to use constant offsets only for *As*

- strided access into *Bs* generates one load and one integer add

Removing the ADD instruction

We store transposed data into *Bs* and modify the inner loop

```
Csub1 += As[ty][k]*Bs[tx][k];
Csub2 += As[ty+16][k]*Bs[tx][k];
```

Removing the ADD instruction

We store transposed data into *Bs* and modify the inner loop

```
Csub1 += As[ty][k]*Bs[tx][k];
Csub2 += As[ty+16][k]*Bs[tx][k];
```

After disassembling, we se there is no ADD instruction

```
...
mov.b32 $r0, s[$ofs4+0x0008]
mad.rn.f32 $r6, s[$ofs3+0x0034], $r0, $r6
mad.rn.f32 $r8, s[$ofs1+0x0008], $r0, $r8
...
```

Removing the ADD instruction

We store transposed data into *Bs* and modify the inner loop

```
Csub1 += As[ty][k]*Bs[tx][k];
Csub2 += As[ty+16][k]*Bs[tx][k];
```

After disassembling, we se there is no ADD instruction

```
...
mov.b32 $r0, s[$ofs4+0x0008]
mad.rn.f32 $r6, s[$ofs3+0x0034], $r0, $r6
mad.rn.f32 $r8, s[$ofs1+0x0008], $r0, $r8
...
```

New issue – memory bank conflicts

Removing the ADD instruction

We store transposed data into *Bs* and modify the inner loop

```
Csub1 += As[ty][k]*Bs[tx][k];
Csub2 += As[ty+16][k]*Bs[tx][k];
```

After disassembling, we se there is no ADD instruction

```
...
mov.b32 $r0, s[$ofs4+0x0008]
mad.rn.f32 $r6, s[$ofs3+0x0034], $r0, $r6
mad.rn.f32 $r8, s[$ofs1+0x0008], $r0, $r8
...
```

New issue – memory bank conflicts

- solved by padding

Removing the ADD instruction

We store transposed data into *Bs* and modify the inner loop

```
Csub1 += As[ty][k]*Bs[tx][k];
Csub2 += As[ty+16][k]*Bs[tx][k];
```

After disassembling, we se there is no ADD instruction

```
...
mov.b32 $r0, s[$ofs4+0x0008]
mad.rn.f32 $r6, s[$ofs3+0x0034], $r0, $r6
mad.rn.f32 $r8, s[$ofs1+0x0008], $r0, $r8
...
```

New issue – memory bank conflicts

- solved by padding

Resulting speed: 276.2 GFlops.

Can we Reach Better Performance?

Our results are petty close to theoretical bound for one load per two MADs.

- to get better performance, tiled algorithm has to be revised

Can we Reach Better Performance?

Our results are petty close to theoretical bound for one load per two MADs.

- to get better performance, tiled algorithm has to be revised

The main issue is that we multiply two tiles in shared memory

- need of usage load instructions together with MAD instructions

Can we Reach Better Performance?

Our results are petty close to theoretical bound for one load per two MADs.

- to get better performance, tiled algorithm has to be revised

The main issue is that we multiply two tiles in shared memory

- need of usage load instructions together with MAD instructions

Can we have only one block in shared memory?

New Tiled Algorithm

We will iteratively perform rank-1 update of tiles in C using column in A and row in B

New Tiled Algorithm

We will iteratively perform rank-1 update of tiles in C using column in A and row in B

- columns in A are read from shared memory

New Tiled Algorithm

We will iteratively perform rank-1 update of tiles in C using column in A and row in B

- columns in A are read from shared memory
- rows in B can be read one after another, so we can use register to do so

New Tiled Algorithm

We will iteratively perform rank-1 update of tiles in C using column in A and row in B

- columns in A are read from shared memory
- rows in B can be read one after another, so we can use register to do so
- tile in C can be stored in registers

New Tiled Algorithm

We will iteratively perform rank-1 update of tiles in C using column in A and row in B

- columns in A are read from shared memory
- rows in B can be read one after another, so we can use register to do so
- tile in C can be stored in registers
- we work in only one operand in shared memory, so explicit loads are not needed

New Tiled Algorithm

We will iteratively perform rank-1 update of tiles in C using column in A and row in B

- columns in A are read from shared memory
- rows in B can be read one after another, so we can use register to do so
- tile in C can be stored in registers
- we work in only one operand in shared memory, so explicit loads are not needed
- theoretical bound is now done by speed of MAD instruction with operand in shared memory: 415 GFlops

New Tiled Algorithm

The best-performing configuration:

- matrix A read by 16×16 tiles, stored in shared memory
- matrix B read by 64×1 tiles, stored in registers
- tiles of matrix C have 64×16 size, they are stored in registers

New Tiled Algorithm

The best-performing configuration:

- matrix A read by 16×16 tiles, stored in shared memory
- matrix B read by 64×1 tiles, stored in registers
- tiles of matrix C have 64×16 size, they are stored in registers

The measured performance is 375 GFlops.

Summary

Implementation	performance	rel. Δ	abs. Δ
Naive, blocks 1×128	3.9 GFlops		
Naive	36.6 GFlops	9.4×	9.4×
Tiled algorithm	198 GFlops	5.4×	51×
Thread blocks 32×16, tiles 32×32	235 GFlops	1.19×	60×
Removing ADD instruction	276 GFlops	1.17×	71×
One block in shared memory	375 GFlops	1.36×	96×

Summary

Implementation	performance	rel. Δ	abs. Δ
Naive, blocks 1×128	3.9 GFlops		
Naive	36.6 GFlops	$9.4\times$	$9.4\times$
Tiled algorithm	198 GFlops	$5.4\times$	$51\times$
Thread blocks 32×16, tiles 32×32	235 GFlops	$1.19\times$	$60\times$
Removing ADD instruction	276 GFlops	$1.17\times$	$71\times$
One block in shared memory	375 GFlops	$1.36\times$	$96\times$

- The most relevant is exploiting memory locality and basic memory access optimization.

Summary

Implementation	performance	rel. Δ	abs. Δ
Naive, blocks 1×128	3.9 GFlops		
Naive	36.6 GFlops	9.4×	9.4×
Tiled algorithm	198 GFlops	5.4×	51×
Thread blocks 32×16, tiles 32×32	235 GFlops	1.19×	60×
Removing ADD instruction	276 GFlops	1.17×	71×
One block in shared memory	375 GFlops	1.36×	96×

- The most relevant is exploiting memory locality and basic memory access optimization.
- Finer optimizations are relatively challenging, important for really performance critical codes.