Double-Precision Matrix Multiply on CUDA

Parallel Computation (CSE 260), Assignment 2

Andrew Conegliano (A53053325)   Matthias Springer (A99500782)   GID G-338-665

February 13, 2014

Assumptions

- All matrices are square matrices and have a size of 256×256, 512×512, 1024×1024, or 2048×2048.
- All matrices consist of double-precision floating point values (64-bit doubles).
- Parameters (grid/block size) and matrix size are chosen in such a way that no fringe cases emerge.

Notation

In this work, we use the following notation and variable names.

- n is the size of one dimension of the matrices involved. I.e., every matrix has n^2 values.
- When referring to matrices, A and B denote the source matrices and C denotes the target matrix, i.e. $C \leftarrow A * B$.
- $i_{B,x}$ and $i_{B,y}$ are the block index in x or y dimension, i.e. `blockIdx.x/y`.
- $dim_{B,x}$ and $dim_{B,y}$ are the block dimension in x or y dimension, i.e. `blockDim.x/y`.
- $i_{T,x}$ and $i_{T,y}$ are the thread index in x or y dimension, i.e. `threadIndex.x/y`.
- $dim_{G,x}$ and $dim_{G,y}$ are the grid dimension in x or y dimension, i.e. `gridDim.x/y`.

1 Runtime Environment

We optimized our implementation for a specific runtime environment. All benchmarks and performance results are based on the following hardware and software.

1.1 Hardware (Dirac)

- 2 Nahalem Quad Core CPUs
- 24 GB DDR3 RAM
- 1 Tesla C2050 (Fermi) GPU
  - 14 vector units $\times$ 32 cores = 448 total cores 1.15 GHz

  - 3 GB DDR5 RAM, 2.625 GB usable due to ECC
  - L2 cache size: 786432 bytes
  - Total constant memory: 65536 bytes
  - Total shared memory per block: 49152 bytes
  - Total number of registers per block: 32768
  - Maximum number of threads per block: 1024
  - Maximum block dimension: 1024 × 1024 × 64
  - Maximum grid dimension: 65535 × 65535 × 65535
  - Warp size: 32
  - Double precision performance peak: 515 Gflops

1.2 Software

- GNU/Linux, kernel version `2.6.32-358.14.1.el6.nersc9.x86_64`
- CUDA 5.5

2 Basic Algorithm

The basic algorithm uses a constant block size and grid dimension of $dim_{G,x} = \lceil \frac{n}{dim_{B,x}} \rceil$ and $dim_{G,y} = \lceil \frac{n}{dim_{B,y}} \rceil$. This ensures that enough threads are spawned, such that every value of C is computed in a separate thread (n^2 threads).

```
c = 0

for k = 0 .. N - 1
    a = A[$i_{B,x}$ * $dim_{B,x} + i_{T,x}$ * N + k]
    b = B[k * N + $i_{B,y}$ * $dim_{B,y} + i_{T,y}$]
    c += a * b

C[($i_{B,x}$ * $dim_{B,x} + i_{T,x}$) * N + $i_{B,y}$ * $dim_{B,y} + i_{T,y}$] = c
```

Figure 1: Naive matrix multiplication algorithm kernel.

Feasible parameter space The basic algorithm has two parameters that we can change: $dim_{B,x}$ and $dim_{B,y}$. A block cannot contain more than 1024 threads, therefore $dim_{B,x} \times dim_{B,y} \leq 1024$. Furthermore, $dim_{B,x}$, $dim_{B,x} < 1024$, because that it maximum block dimension.

Performance Evaluation Figure 2 shows the performance of the basic algorithm at different block sizes (powers of 2) for $n = 256$, $n = 512$, $n = 1024$, and $n = 2048$. For $n = 2048$, we ran the algorithm 5 times and took the average value. For all other matrix sizes, we ran the algorithm 10 times. The white cells in the figure denote invalid combinations of block sizes, i.e. $dim_{B,x} \times dim_{B,y} > 1024$.

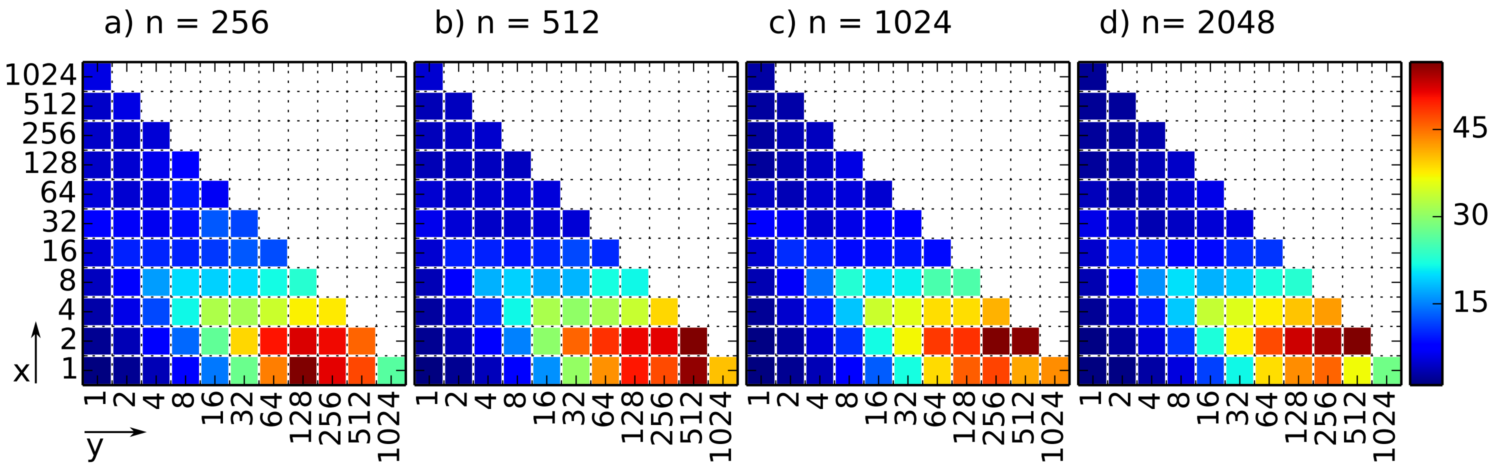

Figure 2: Basic Matrix Multiplication Algorithm at different block sizes.

We reached the following peak performance at certain block sizes.

- $n = 256$: 58.3 GFlops at block size 1×128
- $n = 512$: 54.3 GFlops at block size 2×512
  $n = 512$: 53.4 GFlops at block size 1×512
- $n = 1024$: 56.6 GFlops at block size 2×256
  $n = 1024$: 56.1 GFlops at block size 2×512
- $n = 2048$: 56.5 GFlops at block size 2×512

3 Optimizations

In this section, we present optimizations, that increased the performance of the algorithm beyond 100 GFlops.

3.1 Shared Memory

The first optimization we attempted was to increase the use of shared memory. Shared memory is much faster, at 144 GB/s, than global memory. The basic algorithm makes each thread act independently. By applying the block algorithm, where small blocks of A and B are computed on instead of rows of A and B, we force the use of shared memory, and make threads cooperate. Each thread now loads in a single value of the needed rows/columns of the block into shared memory, then reads from shared memory the values needed for matrix multiplication. This optimization localizes memory accesses to only occur in shared memory, instead of global.

Let us assume that we use a block size of $dim_B \times dim_B$, i.e. we use square block sizes. Then, every element of A and B that is loaded into shared memory, is reused dim_B times, because every element in A or B is used or one row or column of a block in C. Shared memory can, therefore, reduce the global memory traffic.

```
(shared) a[$dim_{B,x}$][$dim_{B,y}$], b[$dim_{B,x}$][$dim_{B,y}$]
c = 0

for k = 0 .. $dim_{G,y}$
     a[$i_{T,x}$][$i_{T,y}$] = A[($i_{B,x}$ * $dim_{B,x}$ + $i_{T,x}$) * N+k * $dim_{B,y}$ + $i_{T,y}$]
     b[$i_{T,y}$][$i_{T,x}$] = B[($i_{B,y}$ * $dim_{B,y}$ + $i_{T,y}$) + N * (k * $dim_{B,x}$ + $i_{T,x}$)]
    (synchronize)

     for kk = 0 .. $dim_{B,x}$
          c += a[kk][$i_{T,x}$] * b[kk][$i_{T,y}$]
    (synchronize)

C[($i_{B,x}$ * $dim_{B,x}$ + $i_{T,x}$)*N + ($i_{B,y}$ * $dim_{B,y}$ + $i_{T,y}$)] = c
```

Figure 3: Shared memory optimization.

Unfortunately, using this optimization barely improved performance because on the Kepler architecture, global memory references are cached.

3.1.1 Implementation Details

Note, that in Figure 3, we need to synchronize all threads after we buffer a block of A and B and after we computed the multiplications, to ensure that the whole block is buffered when the multiplications start and to ensure that no data from the next block is already loaded while some threads still compute multiplications for the previous block.

3.1.2 Parameter Tuning

In this section, the tile size is the number of elements that are cached in the global memory. We show two possible optimizations that we did not implement because of the drawbacks presented in the subsections.

Non-square block sizes We considered the case where block sizes are not square and the tile size equals the block size. Without loss of generality, let x be the bigger dimension, i.e. $dim_{B,x} > dim_{B,y}$. Threads $(dim_{B,y}+1, p), 0 \leq p < dim_{B,y}$ will read values from A and put them into global memory. They will, however, not read the corresponding values[1] $(q, dim_{B,y}+1), 0 \leq q < dim_{B,x}$, because these are not included in the current block. Since we need them for the multiplication, the only way to get them would be from global memory. Therefore, we should always choose square block sizes, if we want to take full advantage of shared memory.

Block size differing from tile size We considered the case where block sizes are square but bigger than the tile size. Since the block size equals the number of threads per block, there will either be threads that do not load any elements from A or B, or data is loaded from A and B multiple times (redundantly). This diminishes the effect of shared memory. Furthermore, elements will have to be loaded into the global memory multiple times. Therefore, if we run out of global memory, it is better to decrease the block size instead of adding another level of blocking inside blocks.

It is, however, possible to run have multiple small tiles per block, effectively resulting in a second level of blocking inside CUDA blocks. This also allows us to use non-square CUDA block sizes. This optimization is discussed in a later subsection.

[1]Coordinates are relative to the current block.

3.1.3 Performance Evaluation

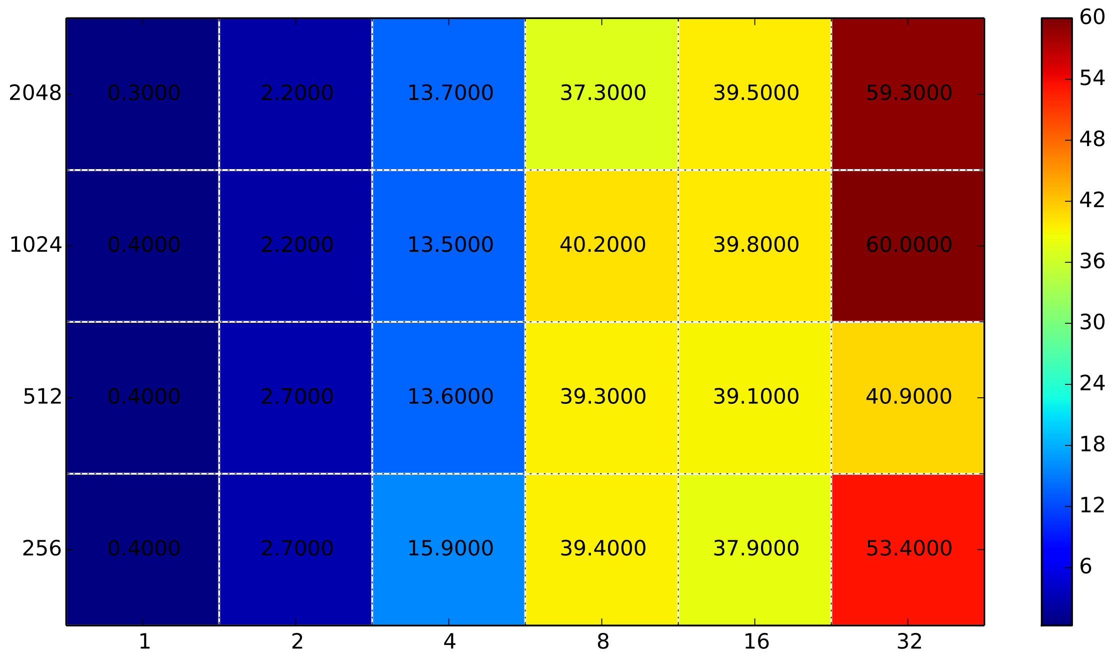

Figure 4: Matrix Multiplication Algorithm using shared memory.

3.2 Memory Coalescing and Memory Banks

In order for faster memory access, coalescing and memory block optimization was implemented. To coalesce global memory basically means to access it in rows sequentially and not columns. We made every thread accesses global memory in row-wise and consecutively in rows by modifying global memory indices. This in turn allows the DRAM on the GPU to consolidate memory requests into a single request, which results in peak bandwidth transfer rate. Also, global memory accesses need to be within a 128 byte segment, and do not need to be in sequential order. Cudamalloc guarantees memory to be aligned by 256 bytes minimum. This means in certain cases, warps will need 2 transactions in order to complete a read instead of 1. To prevent bank conflicts, threads need to access shared memory in columns or in offsets of $16(halfwarp)*32bits$. Every consecutive 32 bits of shared memory is allocated to different bank, max of 32 banks, and half warps are run at a time. Threads are organized by row sequentially, but access shared memory column wise. This allows each thread to access one memory bank, and because only one thread can access a memory bank at a time there are no conflicts.

3.3 Implementation Details

When accessing A and storing it into shared memory, the $k * dim_{B,y}$ allows rows to be loaded and the $i_{T,x}$ are consecutive in value and in the same warp. Each thread accesses elements in the same row, and because all adjacent threads in warps access adjacent elements, the hardware will coalesce the access. The same logic is applied to the B matrix. Memory banks are not conflicted because of the exception of all threads accessing the same 32-bit word, then its a broadcast.

```
(shared) a[$dim_{B,x}$][$dim_{B,y}$], b[$dim_{B,x}$][$dim_{B,y}$]
c = 0

for k = 0 .. $dim_{G,y}$
     a[$i_{T,x}$][$i_{T,y}$] = A[($i_{B,y}$ * $dim_{B,y}$ + $i_{T,y}$) * N+k * $dim_{B,y}$ + $i_{T,x}$]
     b[$i_{T,y}$][$i_{T,x}$] = B[($i_{B,x}$ * $dim_{B,x}$ + $i_{T,x}$) + N * (k * $dim_{B,x}$ + $i_{T,y}$)]
    (synchronize)

     for kk = 0 .. $dim_{B,x}$
          c += a[$i_{T,y}$][kk] * b[kk][$i_{T,x}$]
    (synchronize)

C[($i_{B,y}$ * $dim_{B,x}$ + $i_{T,y}$)*N + ($i_{B,y}$ * $dim_{B,y}$ + $i_{T,y}$)] = c
```

Figure 5: Coalesced memory optimization.

3.3.1 Performance Evaluation

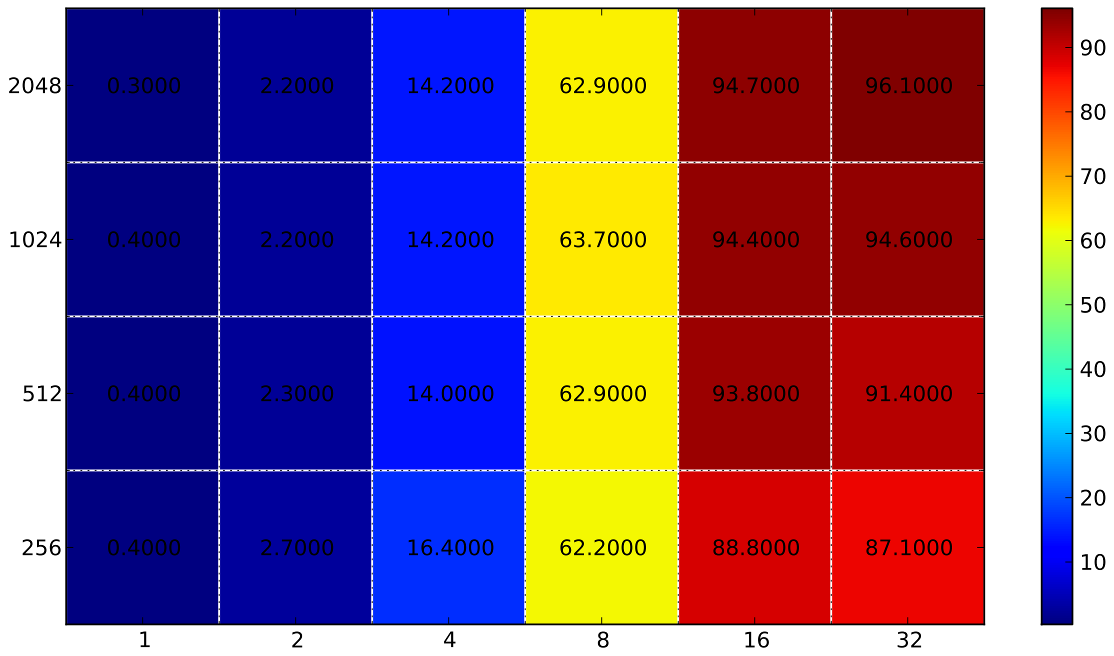

Figure 6: Matrix Multiplication Algorithm using shared memory and coalescing.

3.4 Instruction Level Parallelism

In order to hide memory latency, we chose to increase instruction level parallelism and sacrifice occupancy. The hardware in use needs to keep about 155KB of memory in flight, $500CP(1100nsec) * 144GB/s$, which is met by the amount of threads used. Each thread now has an additional two memory stores and two multiply and add's (MAD), as well as an increase of 4x shared memory. This in turn increased the register use from 22 to 33. Registers are the only way to increase flops because they are the closest memory hierarchy to the hardware, hence the fastest.

3.4.1 Implementation Details

While keeping the block size constant, we reduced the grid dimension by a factor of 2 in both the x and y direction. Therefore, every thread stores 4 values of the original matrix into shared memory, does 4 MAD's, and computes 4 values of C, instead of just 1. A single thread computes the following values of C.

- $\mathrm{C}[i_{B,x} \times dim_{B,x} + i_{T,x}][i_{B,y} \times dim_{B,y} + t_{T,y}]$
- $\mathrm{C}[(i_{B,x} + dim_{G,x}) \times dim_{B,x} + i_{T,x}][i_{B,y} \times dim_{B,y} + t_{T,y}]$
- $\mathrm{C}[i_{B,x} \times dim_{B,x} + i_{T,x}][(i_{B,y} + dim_{G,y}) \times dim_{B,y} + t_{T,y}]$
- $\mathrm{C}[(i_{B,x} + dim_{G,x}) \times dim_{B,x} + i_{T,x}][(i_{B,y} + dim_{G,y}) \times dim_{B,y} + t_{T,y}]$

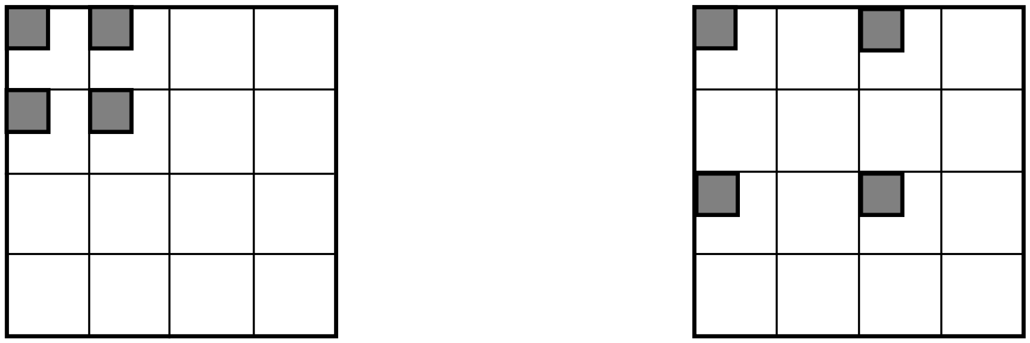
(a) Calculating elements of neighboring blocks. (b) Calculating elements of blocks that are far away.

Figure 7: Comparing access schemes of C when decreasing grid dimension by 2 in both dimensions. Our kernel uses the scheme shown in Figure 7b.

Figure 7 shows two ways of distributing the extra computations that are done by one thread onto to matrix. In Figure 7a, every block computes the values that are contained in the original neighboring block[2]. In Figure 7b, every thread, and therefore every block, computes elements that are far way. The latter scheme proved to be more performant and is used in our final implementation. When the computed elements are far away, it is less likely that threads access the same memory bank, resulting in a higher transfer rate from global memory to shared memory.

[2]Keep in mind that the figure shows only one thread. The thin lines denote the original block boundaries.

3.4.2 Performance Evaluation

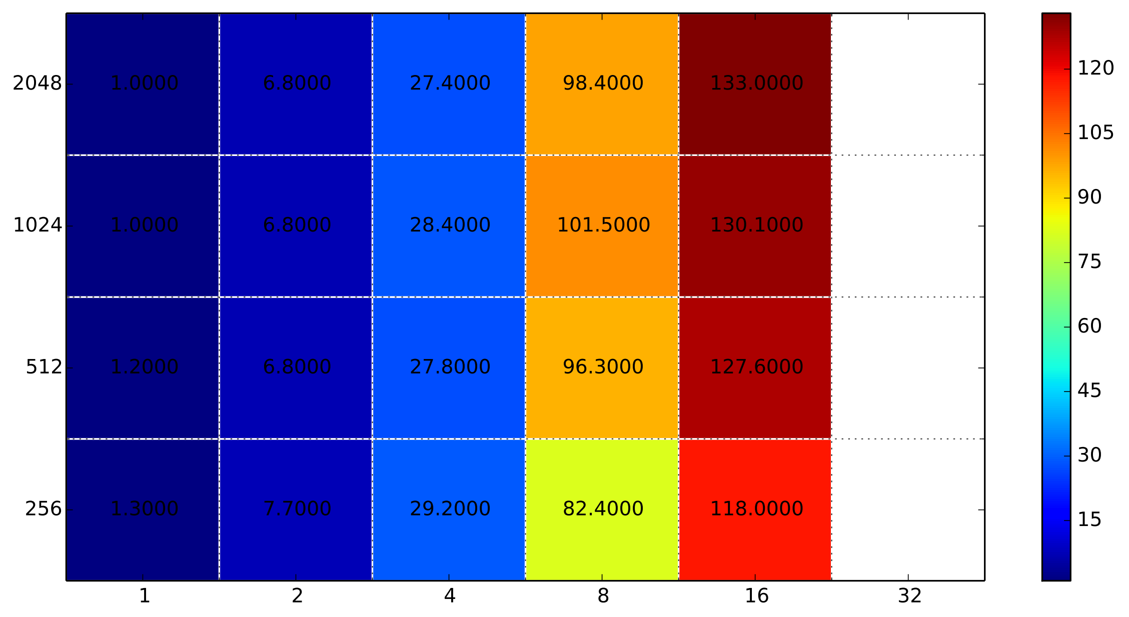

Figure 8: Matrix Multiplication Algorithm using shared memory, coalescing, and ILP.

Figure 8 shows the performance of the our final implementation with all optimizations except for second-level blocking. Note, that a block size of 32 is not feasible, because it exceeds the number of registers per block ($33 \times 32 \times 32 = 33792 > 32768$). Also, due to the unrolling, we use $dim_{B,x} \times dim_{B,y} \times 2$ doubles per matrix A and B, resulting in $dim_{B,x} \times dim_{B,y} \times 32$ bytes, which is 32768 bytes per block for $dim_{B,x} = dim_{B,y} = 32$. This does, however, not exceed the maximum shared memory per block.

3.4.3 Increasing ILP

We also tried to increase ILP further while maintaining square block sizes, by putting even more computations into one thread. This increases the shared memory usage. However, this optimization turned out to be slower. For example, by doing $8 \times 8 = 64$ computations per thread at a block size of 16×16, we utilized $63 \times 16 \times 16 = 16128$ registers per block (out of 32768) and $16 \times 16 \times 2 \times 8 \times 8 = 32786$ bytes shared memory (out of 49152), but it could not make up for the lost occupancy. The performance dropped to 12.5 GFlops at $n = 1024$ and similar values for other matrix sizes.

3.5 Second-level Blocking

In a previous subsection, we explained why the usage of shared memory restricts us to square block sizes. We can circumvent this restriction by adding a second level of blocking per CUDA block. Consider, for instance, that we have a block size of 8×2. When using only one level of blocking, we will eventually run into an *index out of bounds* error, when we try to access the shared memory copy of A or B at the positions $(1, 3), (1, 4), \ldots$, because the algorithm expects a square matrix in the inner `kk` loop (see Figure 5).

Second-level blocking creates smaller blocks inside a CUDA block. In the 8×2 example, we create 4 inner blocks of size 2×2, i.e. the first 4 threads fill a shared memory block, the next 4 threads fill another shared memory block, and so on. These four inner blocks reside inside the same CUDA blocks, but the respective threads do not access shared data from other inner blocks.

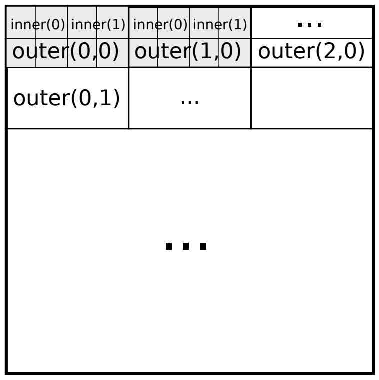

Figure 9: Second-level blocking: Creating inner blocks inside outer (CUDA) blocks.

Figure 9 shows an example for second-level blocking at a CUDA (outer) block size of 8×4. Every inner block has size 4×4 and is, therefore, square.

3.5.1 Implementation Details

We try to keep the number of inner blocks as small as possible. A smaller number of inner blocks results in bigger inner blocks, and a big block size results in more reuse of data, as explained in Section 3.1.

Therefore, we divide a CUDA block of size $dim_{B,x} \times dim_{B,y}$ into inner blocks as follows. Let, without loss of generality, $dim_{B,x} > dim_{B,y}$[3]. Then we have an inner grid dimension of $\frac{dim_{B,x}}{dim_{B,y}} \times 1$. Note, that since we assume that $dim_{B,x}$ and $dim_{B,y}$ divide n evenly, $dim_{B,x}$ is always a multiple of $dim_{B,y}$ for matrix sizes of 256, 512, 1024 and 2048[4]. Every inner block has a size of $dim_{B,y} \times dim_{B,y}$.

3.5.2 Performance Evaluation

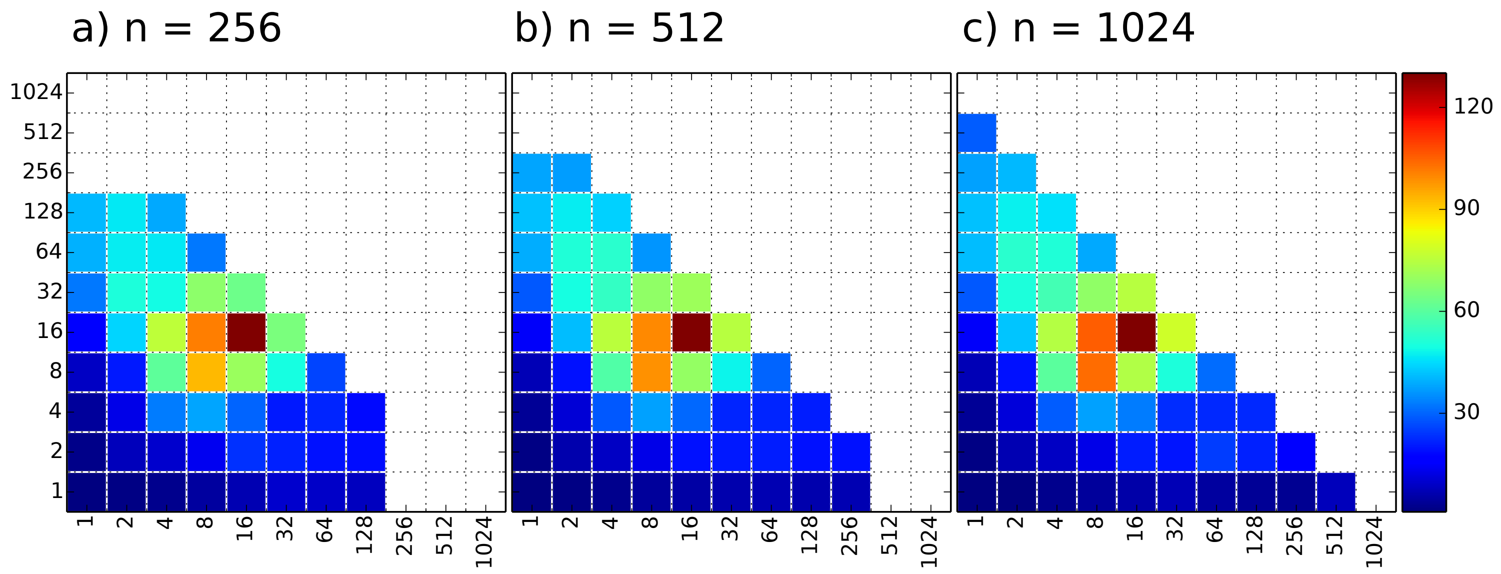

Figure 10: Performance of coalesced, shared-memory algorithm with ILP and second-level blocking at different block sizes.

Figure 10 shows the performance of our final algorithm at different block sizes. For a (CUDA) block size of 16×16, we reach the peak performance (same as in the previous subsection). In this case, we create only

[3]For $dim_{B,x} = dim_{B,y}$, we just create a single inner block.
[4]These are the matrix sizes re ran benchmarks for.

one inner block. This results in the maximum reuse of shared memory, because the inner block is as big as possible. For the next bigger CUDA block size (that devides 256, 512, or 1024 evenly) of 32×32, we run out of registers, thus this configuration is not feasible.

Note, that for $n = 256$, the configurations 1×256 and 256×1 are invalid. The same is the case for $n = 512$ and $n = 1024$. The reason is that our algorithm computes two values of C at a time per thread, due to the ILP optimization presented in a previous subsection. Therefore, $n = 256$ and 1×256 (and the other configurations mentioned) are not feasible, because they leave no space for ILP/unrolling in y direction.

3.6 Further Considerations

Our kernel has no fringe cases because we assume a matrix size evenly divisible by our block sizes. Our kernel also has no thread divergence, because all threads inside one block always follow the same instruction stream. There are no `if` statements that could cause the kernel to diverge.

4 Host CPU computation

Figure 11 shows the performance of the multi-threaded matrix multiplication algorithm that runs on the host CPUs at different matrix sizes and number of threads. For the matrix size, we chose multiples of 32 between 512 and 1504, because we were told that we can reach the peak performance for matrix sizes close to 1024. The results from the last assignment suggest that the peak performance is reached at multiples of 32 (or other powers of 2).

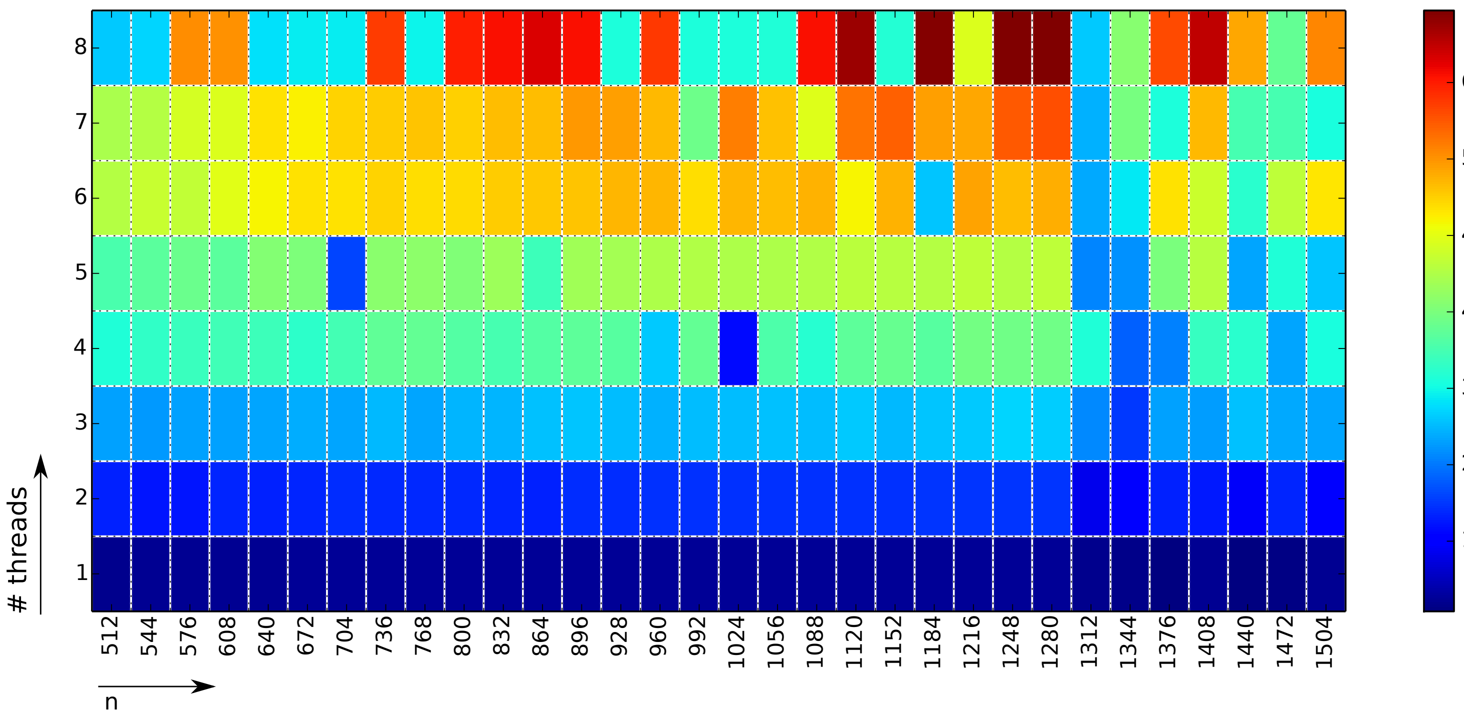

Figure 11: Host CPU multi-threaded performance

For the following parameters, we reached the peak performance. We ran the algorithm for every parameter setting 10 times and took the average value.

- $n = 1120$, 8 threads: 69.95 GFlops
- $n = 1184$, 8 threads: 71.11 GFlops

- $n = 1248$, 8 threads: 71.55 GFlops
- $n = 1280$, 8 threads: 71.54 GFlops

5 Performance Overview

Figure 12 gives an overview of all optimizations and the performance the single-threaded host CPU computation[5]. The CUDA bars show the performance for the best parameters (block size).

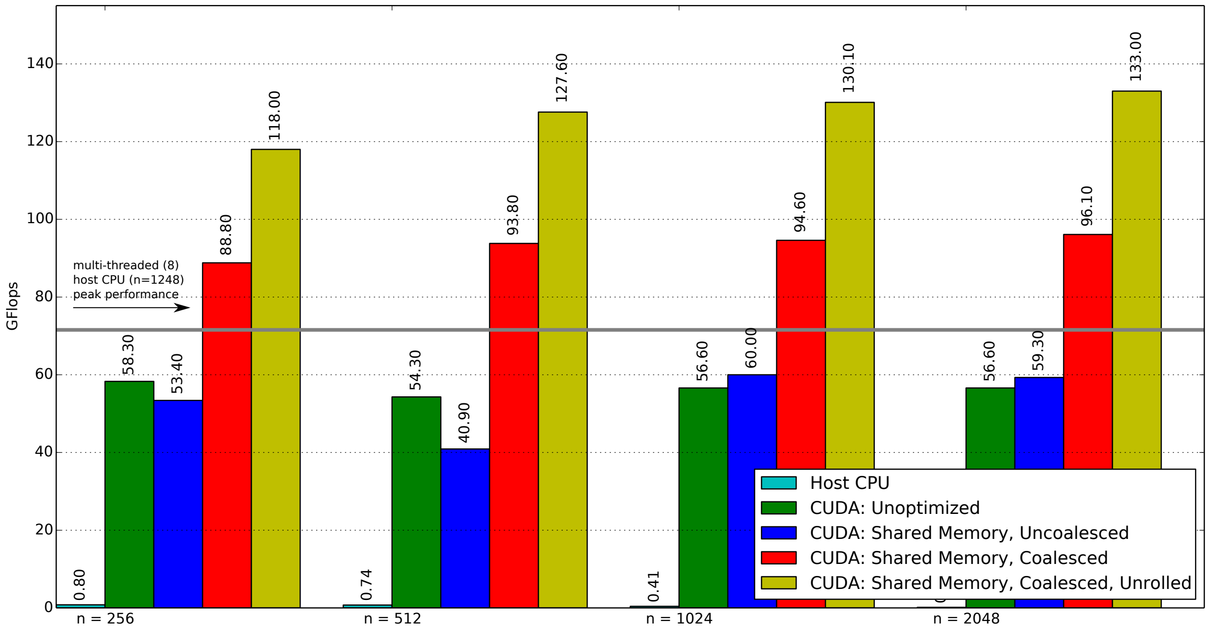

Figure 12: Performance Overview

For the unoptimized single-threaded host CPU computation, we reached the following performance.

- $n = 256$: 0.8 GFlops
- $n = 512$: 0.74 GFlops
- $n = 1024$: 0.41 GFlops
- $n = 2048$: 0.15 GFlops

The performance of the multi-threaded optimized host CPU computation is shown in the previous section for various matrix sizes and number of threads.

[5]This is the performance of the unoptimized host verification code.