8-Bit Integer Matmul on Turing GPUs (Part I)

or: Using Tensorcores without Inline Assembly

Erik Schultheis[1]

February 8, 2025

[1]Aalto University, Helsinki, Finland

Preliminaries

The main goal of this presentation is **educational**

Why 8-bit integer matmul for teaching?

Interesting problem on a wide range of hardware

- AVX: no direct equivalent of FMA
- AVX512-VNNI: `signed` × `unsigned` dot-product instruction
- CUDA (non-TC): `dp4a` instruction
- TensorCore (Turing and newer)

Testability and Determinism

- Results are independent of operation ordering
- No need to calibrate floating-point tolerances.

Goals and non-goals

- Demonstrate basic use of tensor cores directly in CUDA
- Code that works for “nice” matrices: Large, square matrices, size divisible by a large power of 2
- Optimize for `Quadro RTX 4000` (`Turing TU104`)
- No async memcpy, no async mma, no sparsity; `Turing` has less features
  $\Longrightarrow$ less complexity

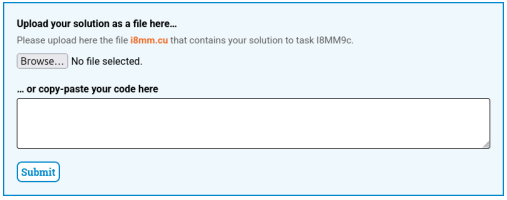
You can try this out!

Task	Attempts	Points	Max	Rating	Rec.	Deadline for full points
I8MM2: CPU baseline Implement a simple **sequential** baseline solution. Do not try to use any form of parallelism yet; try to make it work correctly first.	1	3	3	★		2025-10-08 at 23:59:59
I8MM3: fast CPU Implement a fast CPU solution using multithreading. Use instruction-level parallelism, vectorization, as well as register and cache reuse.	4	5	5	★+		2025-10-22 at 23:59:59
I8MM4: GPU baseline Implement a GPU baseline solution.	1	3	3	★		2025-11-05 at 23:59:59
I8MM5: fast GPU Implement a fast GPU solution using regular (portable) CUDA.	3	3	3	★★		2025-11-19 at 23:59:59
I8MM6: Tensorcore baseline Implement a tensorcore baseline. Do not try to use any forms of data-reuse yet. This is a **technique exercise**, a valid solution **must** make use of the tensor cores.	2	4	4	★★		2025-12-03 at 23:59:59
I8MM9a: CPU AVX512-VNNI Implement a fast CPU solution using AVX512-VNNI instructions.	3	2	4 + 2	★★		2025-12-03 at 23:59:59
I8MM9b: SIMD GPU Implement a fast GPU solution using specialized SIMD instructions available on Turing. This is a **technique exercise**, a valid solution **must** make use of the `__dp4a` intrinsic.	2	4	4 + 2	★★		2025-12-03 at 23:59:59
I8MM9c: fast Tensorcore Implement a fast tensorcore solution.	2	7	7 + 2	★★★		2025-12-03 at 23:59:59

Aalto 2025 Open

ppc-exercises.cs.aalto.fi

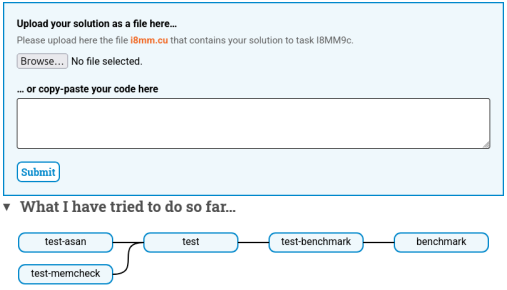
You can try this out!



Aalto 2025 Open

ppc-exercises.cs.aalto.fi

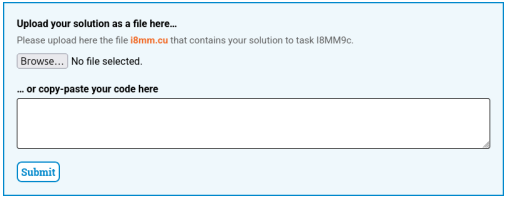
You can try this out!



Aalto 2025 Open

ppc-exercises.cs.aalto.fi

You can try this out!



▼ What I have tried to do so far...

Benchmark	Time	Cycles	Instr.	Threads	Operations	GFLOPS
benchmarks/1a	0.0021 s	2.5×10^6	2.6×10^6	1.00	2.0×10^9	963.2
benchmarks/1b	0.0013 s	3.1×10^6	3.3×10^6	1.00	2.0×10^9	1547.9
benchmarks/2a	0.0096 s	11.4×10^6	23.4×10^6	1.00	432.0×10^9	45210.0
benchmarks/2b	0.1157 s	428.0×10^6	1.0×10^9	1.00	6804.1×10^9	58805.8
benchmarks/3	0.6145 s	2.7×10^9	6.2×10^9	1.00	47775.7×10^9	77742.6

Aalto 2025 Open

ppc-exercises.cs.aalto.fi

How to test custom matmul implementations

Coder's perspective

- Multiply two random matrices, check result. Done?

How to test custom matmul implementations

Coder's perspective

- Multiply two random matrices, check result. Done?

How to test custom matmul implementations

Coder's perspective

- Multiply two random matrices, check result. Done?
- Getting *useful* information out of a test case.

How to test custom matmul implementations

Coder's perspective

- Multiply two random matrices, check result. Done?
- Getting *useful* information out of a test case.
- Small matrices.

How to test custom matmul implementations

Coder's perspective

- Multiply two random matrices, check result. Done?
- Getting *useful* information out of a test case.
- Small matrices.
- Special matrices: large matrix $\times$ identity

How to test custom matmul implementations

Coder's perspective

- Multiply two random matrices, check result. Done?
- Getting *useful* information out of a test case.
- Small matrices.
- Special matrices: large matrix $\times$ identity
- Structured matrices: tiles of constant coefficients

How to test custom matmul implementations

Coder's perspective

- Multiply two random matrices, check result. Done?
- Getting *useful* information out of a test case.
- Small matrices.
- Special matrices: large matrix $\times$ identity
- Structured matrices: tiles of constant coefficients
- Pattern of correct/wrong often more interesting than actual values

How to test custom matmul implementations

Computational perspective

- Calculate reference result with slow (but easily verified) algorithm: difference in speed multiple orders of magnitude!
- Use $C = AB \implies Cx = A(Bx)$, detect errors with *high probability* using only much cheaper matrix-vector multiplications[a]
- Compare against trusted library (cuBLAS)

[a] en.wikipedia.org/wiki/Freivalds%27_algorithm

Tensor core matrix multiplication: The naive implementation

Different views on matrix multiplication

Multiply $A \in \mathcal{I}_8^{m \times k}$ with $B \in \mathcal{I}_8^{k \times n}$ to receive $C \in \mathcal{I}_{32}^{m \times n}$.

Bottom-up

Calculate one element of the result matrix as inner product:

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

Leads to naive $\mathcal{O}(n^k)$ algorithm

Top-down

Large matrix multiplication decomposes into smaller matrix products

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Better decomposition (Strassen's algorithm) can lead to $\mathcal{O}(n^{2.8})$

`https://en.wikipedia.org/wiki/Strassen_algorithm`

The naive algorithm on matrix fragments

Instead of decomposing into a fixed number of submatrices, we can also decompose into fixed-size tiles.

From now on, assume $n, m, k \mod 16 = 0$

Let

$$A = \begin{pmatrix} A_{1:16;1:16} & A_{1:16;17:32} & \dots & A_{1:16;k-16:k} \\ \vdots & \ddots & \ddots & \vdots \\ A_{m-16:m;1:16} & A_{m-16:m;17:32} & \dots & A_{m-16:m;k-16:k} \end{pmatrix} =: \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1k'} \\ \vdots & \ddots & \ddots & \vdots \\ A_{m'1} & A_{m'2} & \dots & A_{m'k'} \end{pmatrix}$$

then

The naive algorithm on matrix fragments

Instead of decomposing into a fixed number of submatrices, we can also decompose into fixed-size tiles.

From now on, assume $n, m, k \mod 16 = 0$

Let

$$A = \begin{pmatrix} A_{1:16;1:16} & A_{1:16;17:32} & \cdots & A_{1:16;k-16:k} \\ \vdots & \ddots & \ddots & \vdots \\ A_{m-16:m;1:16} & A_{m-16:m;17:32} & \cdots & A_{m-16:m;k-16:k} \end{pmatrix} =: \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1k'} \\ \vdots & \ddots & \ddots & \vdots \\ A_{m'1} & A_{m'2} & \cdots & A_{m'k'} \end{pmatrix}$$

then

$$C_{ij} = \sum_{l=1}^{k'} A_{il} B_{lj}$$

The naive algorithm on matrix fragments

Instead of decomposing into a fixed number of submatrices, we can also decompose into fixed-size tiles.

From now on, assume $n, m, k \mod 16 = 0$

Let

$$A = \begin{pmatrix} A_{1:16;1:16} & A_{1:16;17:32} & \cdots & A_{1:16;k-16:k} \\ \vdots & \ddots & \ddots & \vdots \\ A_{m-16:m;1:16} & A_{m-16:m;17:32} & \cdots & A_{m-16:m;k-16:k} \end{pmatrix} =: \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1k'} \\ \vdots & \ddots & \ddots & \vdots \\ A_{m'1} & A_{m'2} & \cdots & A_{m'k'} \end{pmatrix}$$

then

$$C_{ij} = \sum_{l=1}^{k'} \underbrace{A_{il} B_{lj}}_{\text{tensor core}}$$

The CUDA interface to tensor cores

```
template<typename Use, int m, int n, int k, typename T, typename Layout=void> class fragment;

void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm);
void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm, layout_t layout);
void store_matrix_sync(T* mptr, const fragment<...> &a, unsigned ldm, layout_t layout);
void fill_fragment(fragment<...> &a, const T& v);
void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...> &b, const fragment<...> &c, bool satf=false);
```

`fragment`

An overloaded class containing a section of a matrix distributed across all threads in the warp. The mapping of matrix elements into `fragment` internal storage is unspecified and subject to change in future architectures.

Only certain combinations of template arguments are allowed. The first template parameter specifies how the fragment will participate in the matrix operation. Acceptable values for `Use` are:

- `matrix_a` when the fragment is used as the first multiplicand, `A`,
- `matrix_b` when the fragment is used as the second multiplicand, `B`, or
- `accumulator` when the fragment is used as the source or destination accumulators (`C` or `D`, respectively).

The naive algorithm

```
// (i, j) iterate over output elements
// 1 result accumulator per thread
using result_t = int32_t;
using x_fragment_t = int8_t;
using y_fragment_t = int8_t;

result_t v;
v = 0;

for (int l = 0; l < k; l += 1) {
    x_fragment_t x; y_fragment_t y;
    x = A[i * k + l];
    y = B[j + l * n];
    v += x * y;
}

C[j + i * n] = v
```

The naive algorithm

```
// (i, j) iterate over output fragments
// 16 x 16 results per warp => 8 result accumulators per thread
using result_t = fragment<accumulator, 16, 16, 16, int32_t>;
using x_fragment_t = fragment<matrix_a, 16, 16, 16, int8_t, row_major>;
using y_fragment_t = fragment<matrix_b, 16, 16, 16, int8_t, row_major>;

result_t v;
fill_fragment(v, 0);

for (int l = 0; l < k; l += 16) {
    x_fragment_t x; y_fragment_t y;
    load_matrix_sync(x, A + i * 16 * k + l, k);
    load_matrix_sync(y, B + j * 16 + l * n, n);
    mma_sync(v, x, y, v);
}

store_matrix_sync(C + j * 16 + i * n * 16, v, m, mem_col_major);
```

Drumroll...

Drumroll...

▼ All tests passed!

Congratulations, everything looks good, your test has passed all of our tests, and I was also able to run the benchmarks! Here are the running times I got; your code will be graded using **benchmarks/3.txt**:

Benchmark	Time	Cycles	Instr.	Threads	Operations	GFLOPS
benchmarks/1a	0.0018 s	2.1×10^6	3.0×10^6	1.00	2.0×10^9	1133.9
benchmarks/1b	0.0017 s	2.0×10^6	2.8×10^6	1.00	2.0×10^9	1192.7
benchmarks/2	0.1160 s	353.4×10^6	854.9×10^6	1.00	432.0×10^9	3723.3
benchmarks/3	**1.5199 s**	6.7×10^9	15.5×10^9	1.00	6804.1×10^9	4476.7

Drumroll...

▼ All tests passed!

Congratulations, everything looks good, your test has passed all of our tests, and I was also able to run the benchmarks! Here are the running times I got; your code will be graded using **benchmarks/3.txt**:

Benchmark	Time	Cycles	Instr.	Threads	Operations	GFLOPS
benchmarks/1a	0.0018 s	2.1×10^6	3.0×10^6	1.00	2.0×10^9	1133.9
benchmarks/1b	0.0017 s	2.0×10^6	2.8×10^6	1.00	2.0×10^9	1192.7
benchmarks/2	0.1160 s	353.4×10^6	854.9×10^6	1.00	432.0×10^9	3723.3
benchmarks/3	1.5199 s	6.7×10^9	15.5×10^9	1.00	6804.1×10^9	4476.7

4000 billion operations per second sounds like a lot...
but Speed of Light is ≈ 114TOPS

SPECIFICATIONS

GPU Memory	8 GB GDDR6
Memory Interface	256-bit
Memory Bandwidth	Up to 416 GB/s
NVIDIA CUDA® Cores	2304
NVIDIA Tensor Cores	288
NVIDIA RT Cores	36
Single-Precision Performance	7.1 TFLOPS
Tensor Performance	57.0 TFLOPS
System Interface	PCI Express 3.0 x16
Power Consumption	Total board power: 160 W Total graphics power: 125 W
Thermal Solution	Active
Form Factor	4.4" H x 9.5" L, Single Slot
Max Simultaneous Displays	4x 3840x2160 @ 120 Hz 4x 5120x2880 @ 60 Hz 2x 7680x4320 @ 60 Hz
VR Ready	Yes

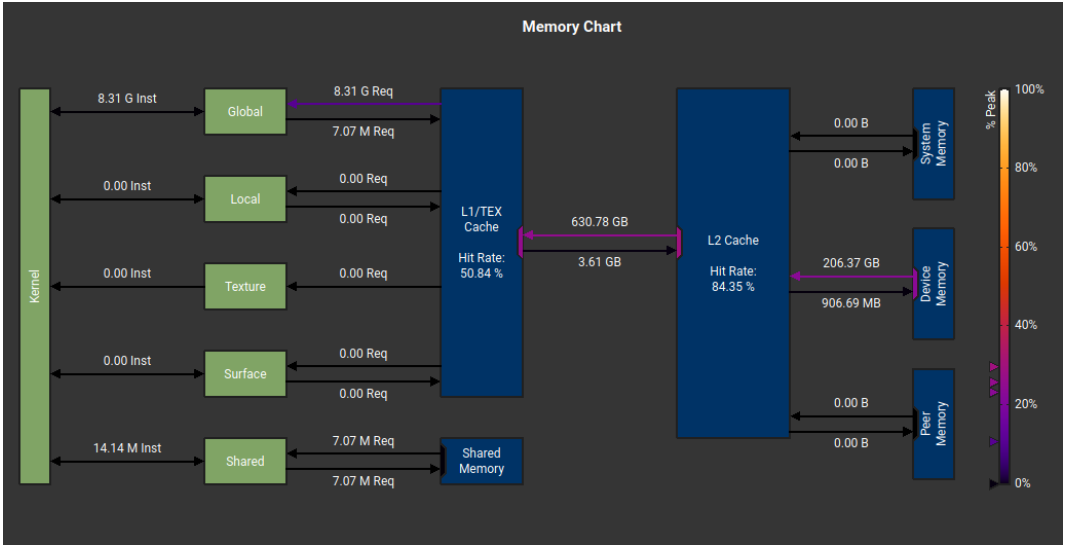

NVIDIA
WHERE'S MY DATA?
WHY
HOW GPU COMPUTING WORKS
Stephen Jones, GTC 2021
Pause (k)

Bad memory access pattern within tensorcore instructions

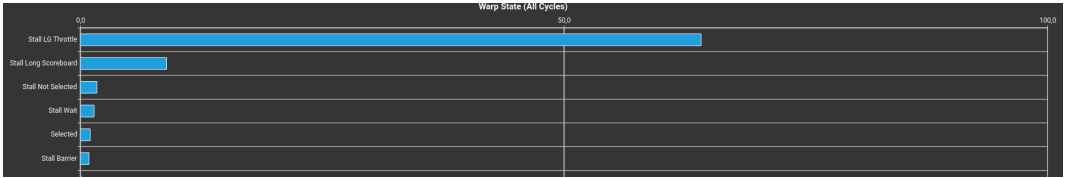
- `load_matrix_sync` fetches from *global* memory and distributes data across threads in the warp, storing parts of the input in individual threads' *registers*.
- Access across the k dimension of matrix B is strided.
- $\implies$ uncoalesced reads.

```
using y_fragment_t = fragment<matrix_b, 16, 16, 16, int8_t, row_major>;
// ...
load_matrix_sync(y, B + j * 16 + l * n, n);
```

Bad memory access pattern within tensorcore instructions



Bad memory access pattern within tensorcore instructions



Uncoalesced access

$\Longrightarrow$ queue of outstanding loads fills up

$\Longrightarrow$ warp stalls when trying to issue new load

Fixing memory access pattern by transposition

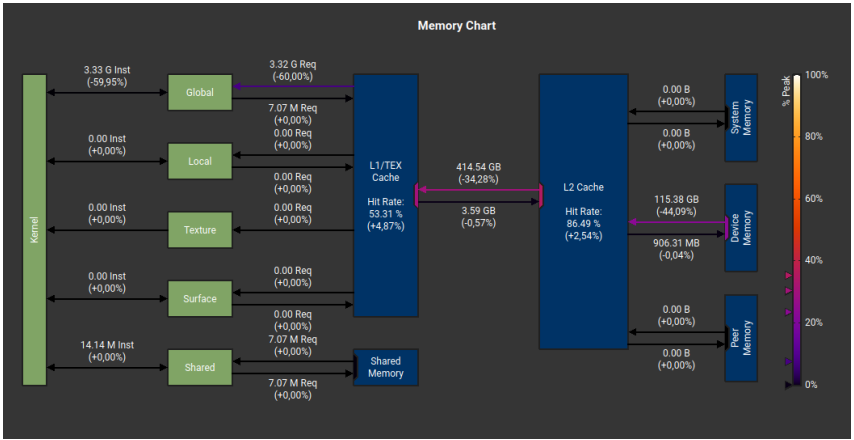
```
// (i, j) iterate over output fragments
// 16 x 16 results per warp => 8 result accumulators per thread
using result_t = fragment<accumulator, 16, 16, 16, int32_t>;
using x_fragment_t = fragment<matrix_a, 16, 16, 16, int8_t, row_major>;
using y_fragment_t = fragment<matrix_b, 16, 16, 16, int8_t, col_major>;

result_t v;
fill_fragment(v, 0);

for (int l = 0; l < k; l += 16) {
    x_fragment_t x; y_fragment_t y;
    load_matrix_sync(x, A + i * 16 * k + l, k);
    load_matrix_sync(y, B + j * 16 * k + l, k);
    mma_sync(v, x, y, v);
}

store_matrix_sync(C + j * 16 + i * n * 16, v, m, mem_col_major);
```

How to get the transposed matrix $B^\top$?

Just run a separate kernel before the matmul. $\mathcal{O}(n^2)$ vs $\mathcal{O}(n^3)$.
$\implies$ negligible overhead:

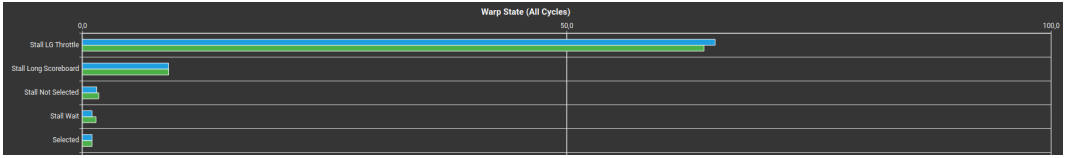
n=15040	row-major	col-major
preproc	–	6.198 ms
gemm	1482.537 ms	922.197 ms

Memory view for the improved kernel

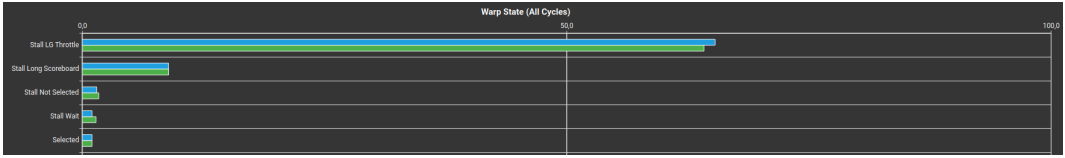


Fixing access pattern reduces amount of memory that needs to be transferred!

Does this fix the LG-Throttle stalls?

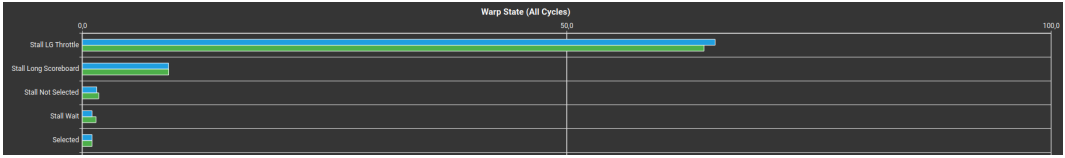


Does this fix the LG-Throttle stalls?

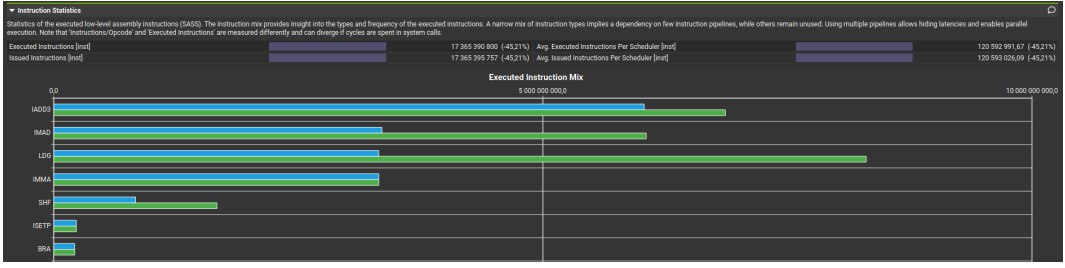


No? Why?

Does this fix the LG-Throttle stalls?



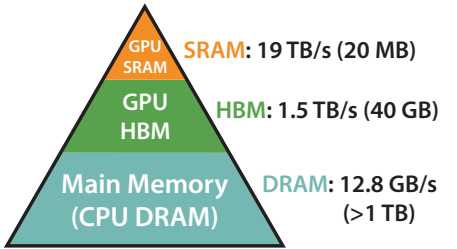
No? Why?



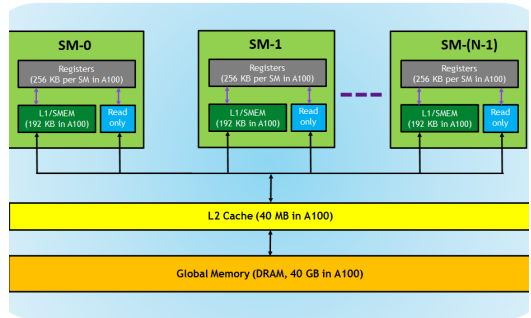
Stalls are cycles spent in state per Selected cycle. We need far fewer instructions, so fewer Selected cycles.

We need to get out our standard kernel optimization toolkit

Step 1: Memory reuse



Memory Hierarchy with Bandwidth & Memory Size



T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness(2022). arXiv preprint arXiv:2205.14135.

`https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/`

Start by reusing the *fastest* memory

Instead of calculating a single fragment per warp (8 outputs per thread), each warp handles 3×3 fragments at once.

$\implies$ Loading 6 input fragments for 9 output fragments;

"arithmetic density" $3/2$, vs $1/2$ for naive kernel.

Start by reusing the *fastest* memory

Instead of calculating a single fragment per warp (8 outputs per thread), each warp handles 3×3 fragments at once.

$\implies$ Loading 6 input fragments for 9 output fragments;

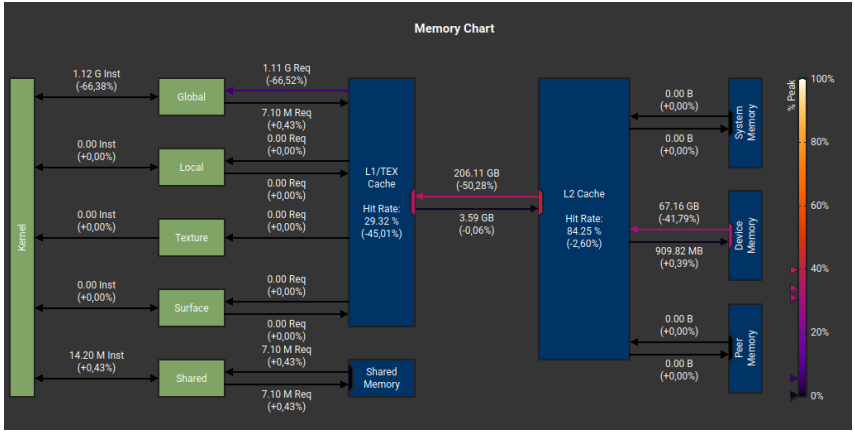
"arithmetic density" $3/2$, vs $1/2$ for naive kernel.

Exactly the same procedure as when optimizing a scalar matmul kernel!

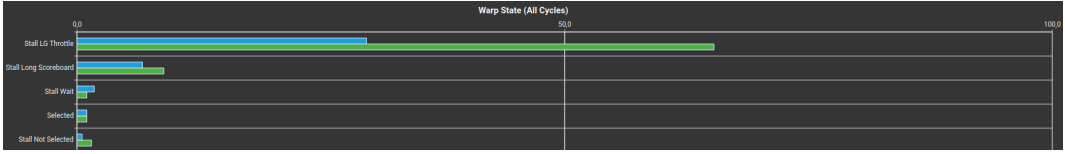
3×3 tile per warp

```
result_t v[3][3];
// ... initialize to zero

for (int l = 0; l < k; l += 16) {
    y_fragment_t y_fragments[3];
    for (int jj = 0; jj < 3; ++jj)
        load_matrix_sync(y_fragments[jj], B + (j + jj) * 16 * k + l, k);

    for (int ii = 0; ii < 3; ++ii) {
        x_fragment_t x;
        load_matrix_sync(x, A + (i + ii) * 16 * k + l, k);
        for (int jj = 0; jj < 3; ++jj) {
            mma_sync(v[jj][ii], x, y_fragments[jj], v[jj][ii]);
        }
    }
}
```

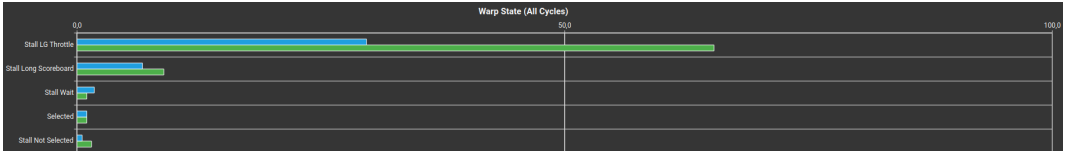
Register reuse drastically reduces memory transfer



Finally, a reduction in throttle stalls

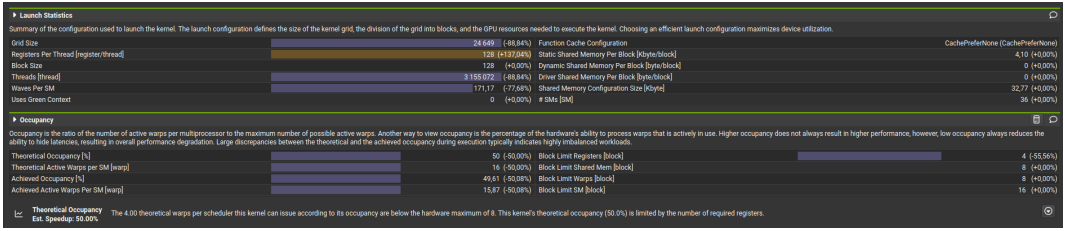


Finally, a reduction in throttle stalls



Any drawbacks?

Register pressure limits occupancy



Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	24 649	(-88,84%)	Function Cache Configuration	CachePreferNone	(CachePreferNone)
Registers Per Thread [register/thread]	128	(+137,04%)	Static Shared Memory Per Block [kbyte/block]	4,10	(+0,00%)
Block Size	128	(+0,00%)	Dynamic Shared Memory Per Block [byte/block]	0	(+0,00%)
Threads [thread]	3 155 072	(-88,84%)	Driver Shared Memory Per Block [byte/block]	0	(+0,00%)
Waves Per SM	171,17	(-77,68%)	Shared Memory Configuration Size [Kbyte]	32,77	(+0,00%)
Uses Green Context	0	(+0,00%)	# SMs [SM]	36	(+0,00%)

Occupancy

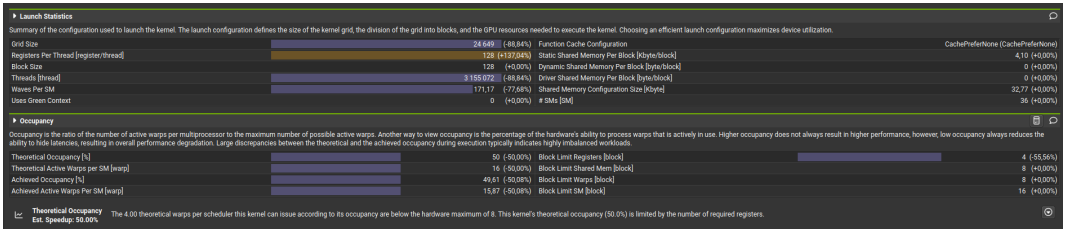
Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	50	(-50,00%)	Block Limit Registers [block]	4	(-55,56%)
Theoretical Active Warps per SM [warp]	16	(-50,00%)	Block Limit Shared Mem [block]	8	(+0,00%)
Achieved Occupancy [%]	49,61	(-50,08%)	Block Limit Warps [block]	8	(+0,00%)
Achieved Active Warps Per SM [warp]	15,87	(-50,08%)	Block Limit SM [block]	16	(+0,00%)

Theoretical Occupancy
Est. Speedup: 50.00%
The 4.00 theoretical warps per scheduler this kernel can issue according to its occupancy are below the hardware maximum of 8. This kernel's theoretical occupancy (50.0%) is limited by the number of required registers.

$\Longrightarrow$ Less effective automatic latency hiding, but much less latency to hide

Register pressure limits occupancy



Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	24 649	(-88,84%)	Function Cache Configuration	CachePreferNone	(CachePreferNone)
Registers Per Thread [register/thread]	128	(+137,04%)	Static Shared Memory Per Block [kbyte/block]	4,10	(+0,00%)
Block Size	128	(+0,00%)	Dynamic Shared Memory Per Block [byte/block]	0	(+0,00%)
Threads [thread]	3 155 072	(-88,84%)	Driver Shared Memory Per Block [byte/block]	0	(+0,00%)
Waves Per SM	171,17	(-77,68%)	Shared Memory Configuration Size [Kbyte]	32,77	(+0,00%)
Uses Green Context	0	(+0,00%)	# SMs [SM]	36	(+0,00%)

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	50	(-50,00%)	Block Limit Registers [block]	4	(-55,56%)
Theoretical Active Warps per SM [warp]	16	(-50,00%)	Block Limit Shared Mem [block]	8	(+0,00%)
Achieved Occupancy [%]	49,61	(-50,08%)	Block Limit Warps [block]	8	(+0,00%)
Achieved Active Warps Per SM [warp]	15,87	(-50,08%)	Block Limit SM [block]	16	(+0,00%)

Theoretical Occupancy
Est. Speedup: 50.00%
The 4.00 theoretical warps per scheduler this kernel can issue according to its occupancy are below the hardware maximum of 8. This kernel's theoretical occupancy (50.0%) is limited by the number of required registers.

$\Longrightarrow$ Less effective automatic latency hiding, but much less latency to hide

$\Longrightarrow$ Limited further improvements through register reuse. But there is one more type of memory that is currently unused...

Shared memory as a “fast” cache

Reuse data *across* warps by sharing input matrices in `__shared__` memory.
For simplicity: 9 warps per block, 3×3 warp-tiles per block: each warp loads one 3×3 fragments tile.
(In total, each block handles tiles of $3 \times 3 \times 16 = 144$ elements)

Loading data into shared memory

Use *vectorized* loads to ensure maximum transfer speed.

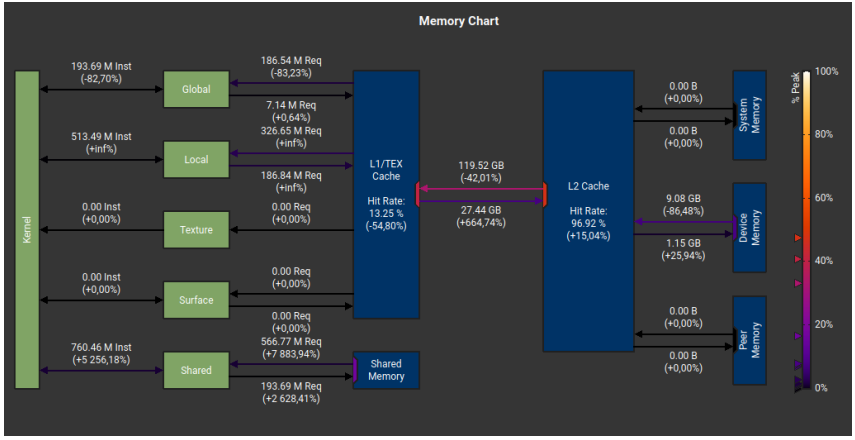
```
// prologue
int lane_id = threadIdx.x % 32;
int warp_id = threadIdx.x / 32;
 __shared__ int8_t xx[3*3][16*16];

// ... in the main loop
for (int ks = 0; ks < k; ks += K) {
    __syncthreads();
    const int8_t *x_ptr = A + l + (ib + warp_id) * 16 * k;
    int s = (32 * lane_id) % 16;
    for (int t = (32 * lane_id) / 16; t < 16; t += (32 * 32) / 16) {
        int4 l = *reinterpret_cast<const int4 *>(x_ptr + t * k + s);
        *reinterpret_cast<int4 *>(&xx[warp_id][t * 16 + s]) = l;
    }
    // ...
```

Reading matrices from shared memory

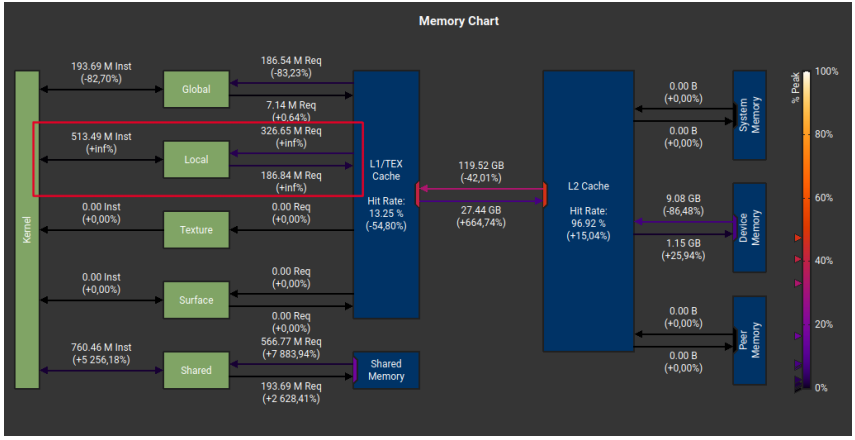
Calculations same as before, only `ldd` (leading dimension) is now 16.

```
__syncthreads();

y_fragment_t y_fragments[3];
for (int jj = 0; jj < 3; ++jj) {
    load_matrix_sync(y_fragments[jj], &yy[jj + 3 * js][0], 16);
}

for (int ii = 0; ii < 3; ++ii) {
    x_fragment_t x;
    load_matrix_sync(x, &xx[ii + 3 * is][0], 16);
    for (int jj = 0; jj < 3; ++jj) {
        mma_sync(v[jj][ii], x, y_fragments[jj], v[jj][ii]);
    }
}
```

Shared memory reuse drastically reduces memory transfer

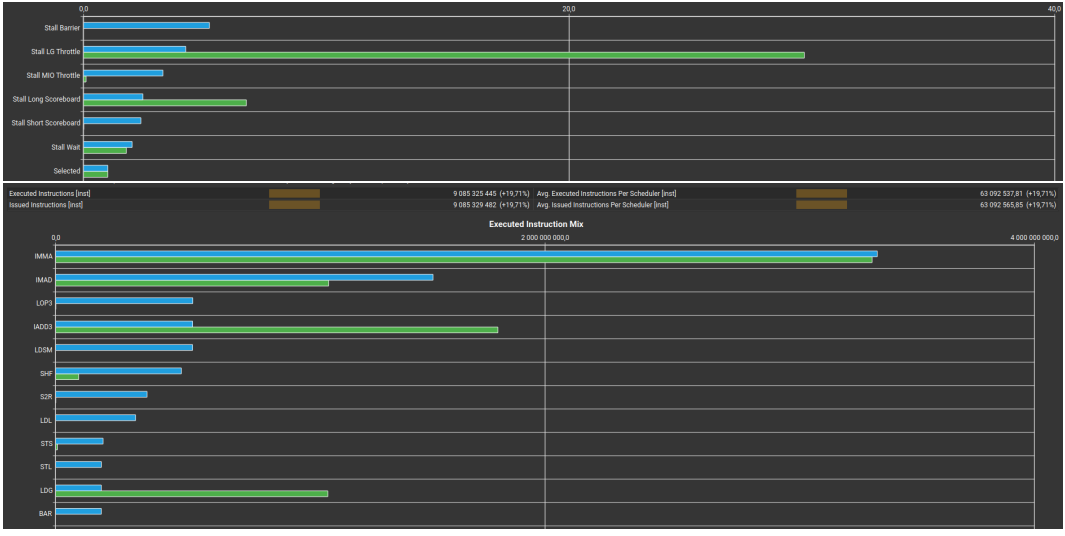


Shared memory reuse drastically reduces memory transfer



Request at least 2 concurrent blocks to hide barriers $\Longrightarrow$ register spilling

Throttle stalls are almost eliminated



Taking stock

Just applying standard optimizations, we have achieved a $4\times$ speedup:

n=15040	row-major	col-major	register reuse	shared memory
preproc	–	6.2 ms	6.3 ms	6.1 ms
gemm	1482.5 ms	922.2 ms	513.2 ms	331.9 ms

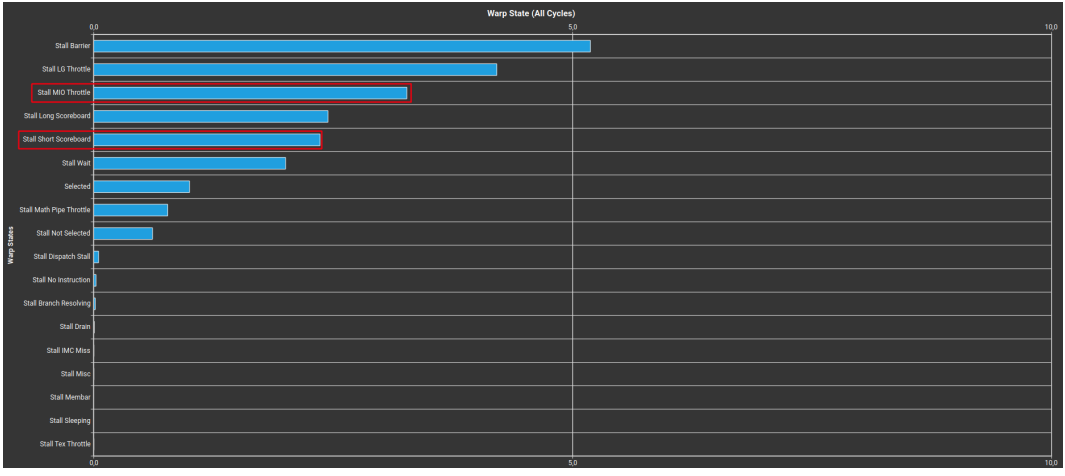
Taking stock

Just applying standard optimizations, we have achieved a $4\times$ speedup:

n=15040	row-major	col-major	register reuse	shared memory
preproc	–	6.2 ms	6.3 ms	6.1 ms
gemm	1482.5 ms	922.2 ms	513.2 ms	331.9 ms

What are the new bottlenecks

Shared memory is not actually all that fast



Step 2: Data layout

Avoiding data shuffling during the critical load phase

`load_matrix_sync` maps input matrix data to registers for *each* load operation reading from shared memory. Can we avoid that?

Avoiding data shuffling during the critical load phase

`load_matrix_sync` maps input matrix data to registers for *each* load operation reading from shared memory. Can we avoid that?
“The mapping of matrix elements into fragment internal storage is unspecified and subject to change in future architectures.”

Avoiding data shuffling during the critical load phase

`load_matrix_sync` maps input matrix data to registers for *each* load operation reading from shared memory. Can we avoid that?

"The mapping of matrix elements into fragment internal storage is unspecified and subject to change in future architectures."

Reasonable assumption: The layout does not change during the runtime of the program :)

$\Longrightarrow$ Shuffle data once using `load_matrix_sync`, then operate on the resulting layout.

Transferring unspecified layout

```
template <class Fragment>
__device__ void mma_store_direct(const Fragment &frag, Fragment *target) {
    int lane_id = threadIdx.x % 32;
    reinterpret_cast<int2 *>(target)[lane_id] =
        *reinterpret_cast<const int2 *>(frag.x);
    __syncwarp();
}

template <class Fragment>
__device__ Fragment mma_load_direct(const Fragment *source) {
    int lane_id = threadIdx.x % 32;
    Fragment frag;
    *reinterpret_cast<int2 *>(frag.x) =
        reinterpret_cast<const int2 *>(source)[lane_id];
    __syncwarp();
    return frag;
}
```

Updated loading procedure...

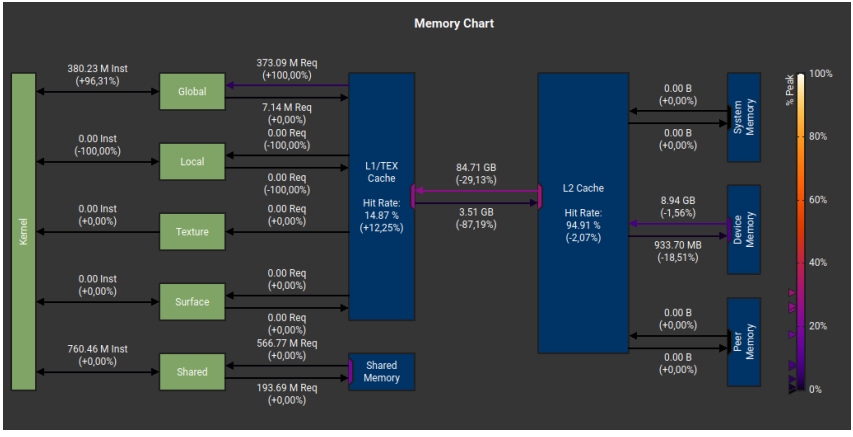
```
__syncthreads();
x_fragment_t x;
load_matrix_sync(x, A + l + (ib + warp_id) * 16 * k, k);
mma_store_direct(x, &xx[warp_id][0]);

y_fragment_t y;
wmma::load_matrix_sync(y, B + l + (jb + warp_id) * k * 16, k);
mma_store_direct(y, &yy[warp_id][0]);
__syncthreads();
```

… and corresponding computation

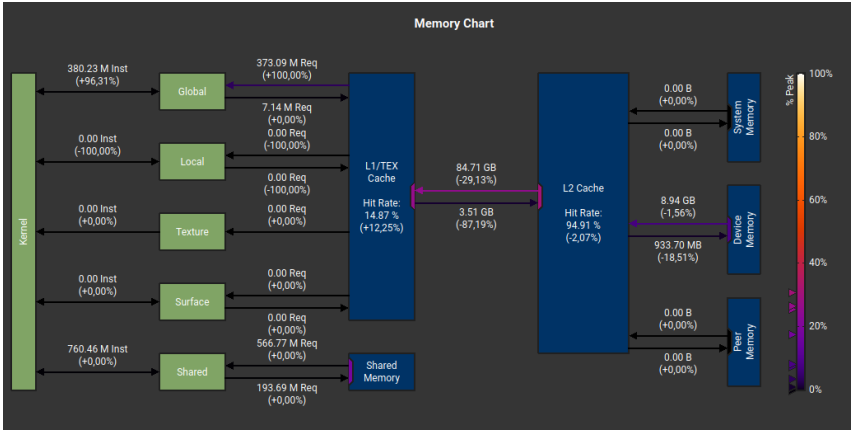
```
y_fragment_t y_fragments[3];
for (int jj = 0; jj < 3; ++jj) {
    y_fragments[jj] = mma_load_direct(&yy[jj + 3 * js][0]);
}

for (int ii = 0; ii < 3; ++ii) {
    x_fragment_t x = mma_load_direct(&xx[ii + 3 * is][0]);
    for (int jj = 0; jj < 3; ++jj) {
        mma_sync(v[jj][ii], x, y_fragments[jj], v[jj][ii]);
    }
}
```

Innermost loop now uses vectorized load instructions!

Trade-off shared memory access and global memory access

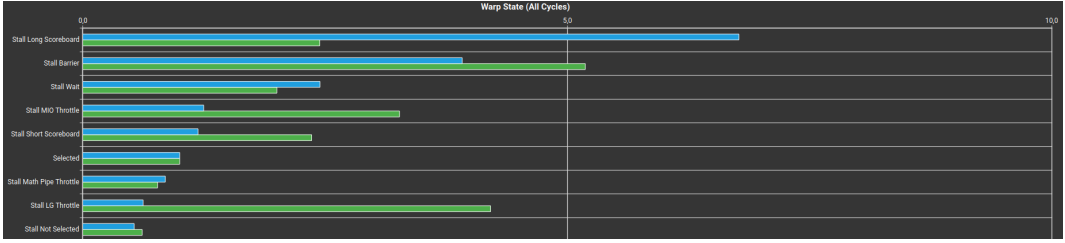


Trade-off shared memory access and global memory access

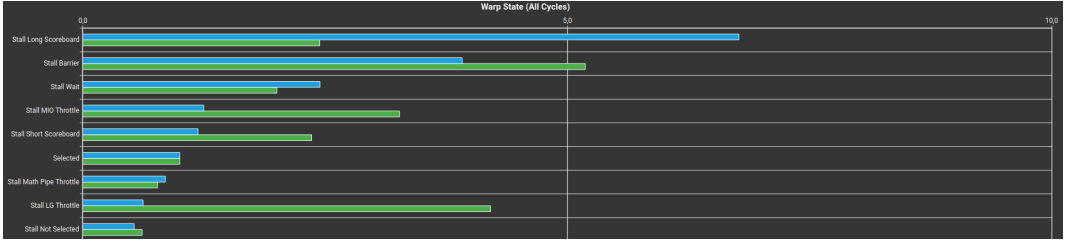


- No more register spills.
- Why do we have more global memory access?

Trade-off shared memory access and global memory access



Trade-off shared memory access and global memory access



- Short scoreboard stalls reduced
- Long scoreboard stalls increased drastically
- LG Throttle reduced drastically (register spilling?)
- Overall speed-up only $\approx 7\%$

Lift data shuffling completely out of the main GEMM kernel

Similar to transpose, pre-shuffling is a $\mathcal{O}(n^2)$ operation.

Lift data shuffling completely out of the main GEMM kernel

Similar to transpose, pre-shuffling is a $\mathcal{O}(n^2)$ operation.

```
__global__ void swizzle_kernel_a(x_fragment_t *a,
                                 const int8_t *a_raw, int m, int k) {
    int warp_id = threadIdx.x / 32;
    int i = 2 * blockIdx.x + warp_id % 2;
    int j = 2 * blockIdx.y + warp_id / 2;

    if (i * 16 >= m || j * 16 >= k)
        return;

    x_fragment_t x;
    load_matrix_sync(x, a_raw + (j + i * k) * 16, k);
    mma_store_direct(x, a + (j + i * k / 16) * 32);
}
```

Back to vectorized loads from global memory

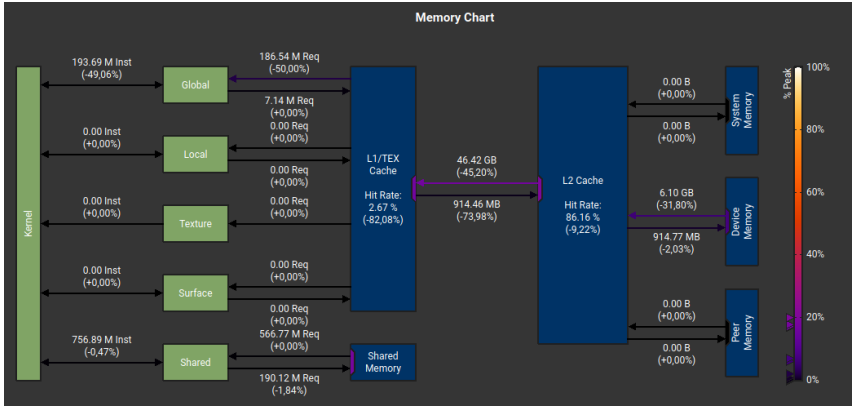
```
__syncthreads();

x_fragment_t x = mma_load_direct(A + (l/16 + (ib + warp_id) * k/16) * 32);
mma_store_direct(x, &xx[warp_id][0]);

y_fragment_t y = mma_load_direct(B + (l/16 + (jb + warp_id) * k/16) * 32);
mma_store_direct(y, &yy[warp_id][0]);

__syncthreads();
```

Back to vectorized loads from global memory

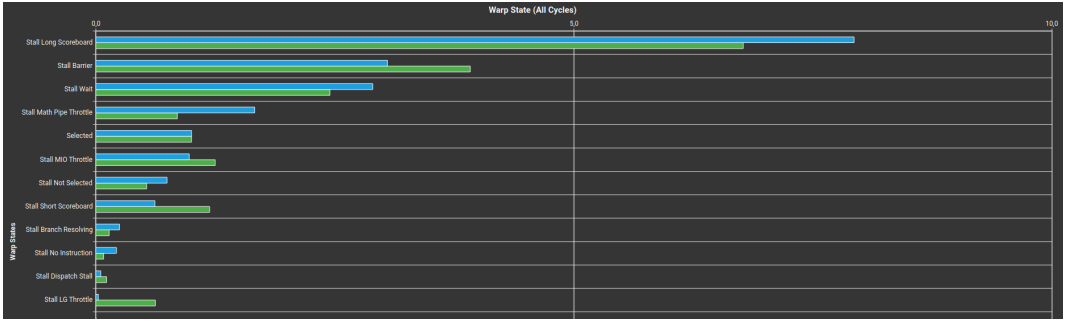
```
__syncthreads();

x_fragment_t x = mma_load_direct(A + (l/16 + (ib + warp_id) * k/16) * 32);
mma_store_direct(x, &xx[warp_id][0]);

y_fragment_t y = mma_load_direct(B + (l/16 + (jb + warp_id) * k/16) * 32);
mma_store_direct(y, &yy[warp_id][0]);

__syncthreads();
```

Computation loop remains unchanged.

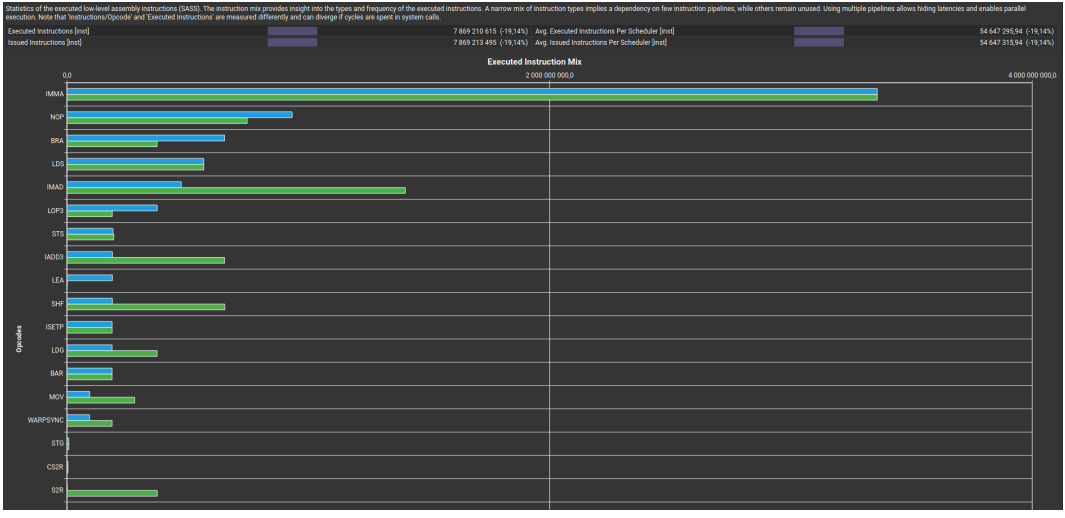
Global memory access is back to old values



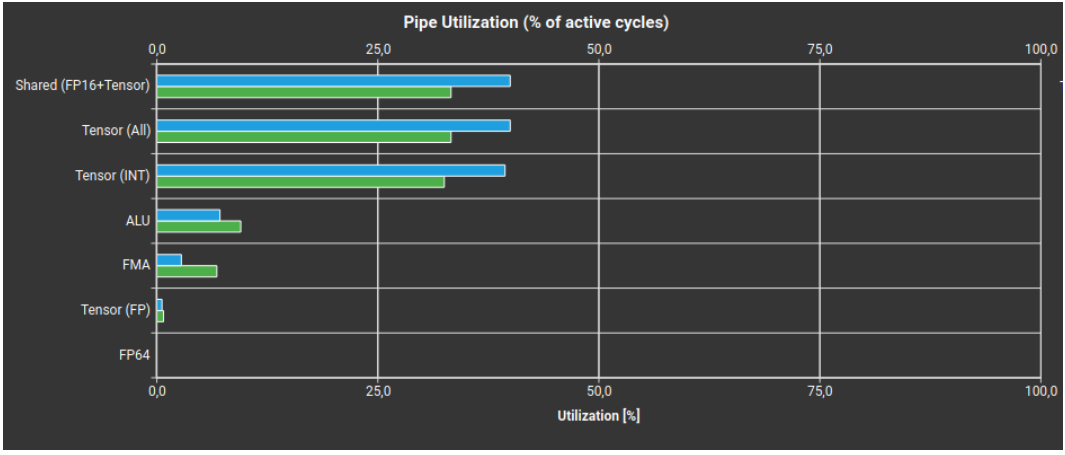
Warp Stalls


Warp State (All Cycles)
0,0
5,0
10,0
Warp States
Stall Long Scoreboard
Stall Barrier
Stall Wait
Stall Math Pipe Throttle
Selected
Stall MIO Throttle
Stall Not Selected
Stall Short Scoreboard
Stall Branch Resolving
Stall No Instruction
Stall Dispatch Stall
Stall LG Throttle

Reduced instructions due to linear loads



Where are we compared to Speed-of-Light?



About 40% utilization of tensor cores.

Validating the pre-processing trade-off

n=15040	row-major	col-major	registers	smem	shuffle 1	shuffle 2
preproc	–	6.2 ms	6.3 ms	6.1 ms	5.1 ms	11.2 ms
gemm	1482.5 ms	922.2 ms	513.2 ms	331.9 ms	193.8 ms	178.3 ms

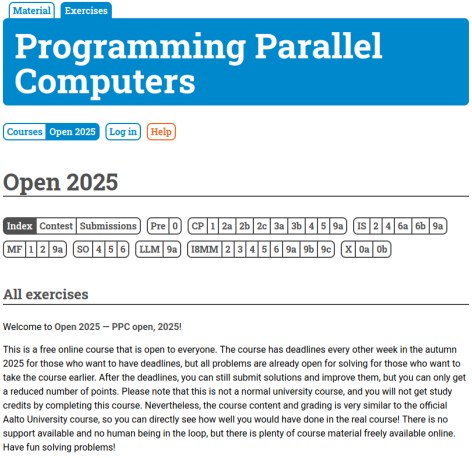
Faster main kernel compensates for doubling of pre-processing time.

Outlook

Further Optimizations (if interested in Part II):

- Double buffering
- Large block sizes
- Full vectorization
- Careful instruction sequencing

Try it out!



Aalto 2025 Open

ppc-exercises.cs.aalto.fi

questions, suggestions, bugs: feel free to ping me on GPUMode @ngc92.