Matrix Extension Modelling

Miles Rusch, Dongjie Xie, Alex Rucker

Tenstorrent

Overview

Introduction to Matrix Multiplication (GEMM) Kernels

- Overview of BLAS decomposition of large GEMM into outer product micro-kernels

Outer Product Instruction Extension

- Matrix ISA extension targets outer product GEMM kernels
- Functional unit layout

Modelling Resources and Performance

- **Model Inputs:** datatype, memory latency, matrix dimensions, functional unit size, etc
- **Estimated Resource Requirements:** memory bandwidth, instruction decode bandwidth, matrix register file (MRF) capacity, multiply-accumulate (MACC) logic size
- **Estimated Performance Metrics:** Operations per Cycle (OPC), Kernel Latency, MMU Utilization

Extending Ocelot

- RTL Implementation of RISC-V Vector Extension (RVV)
- Extending RVV with Matrix ISA

GEMM

- Often Compute Bound:
  - $\sim N^2$ Memory Ops
  - $\sim N^3$ MACC Ops
- Many vector schedules exist to accelerate GEMM (eg Fig. 1)
- Matrix Operations can further accelerate GEMM

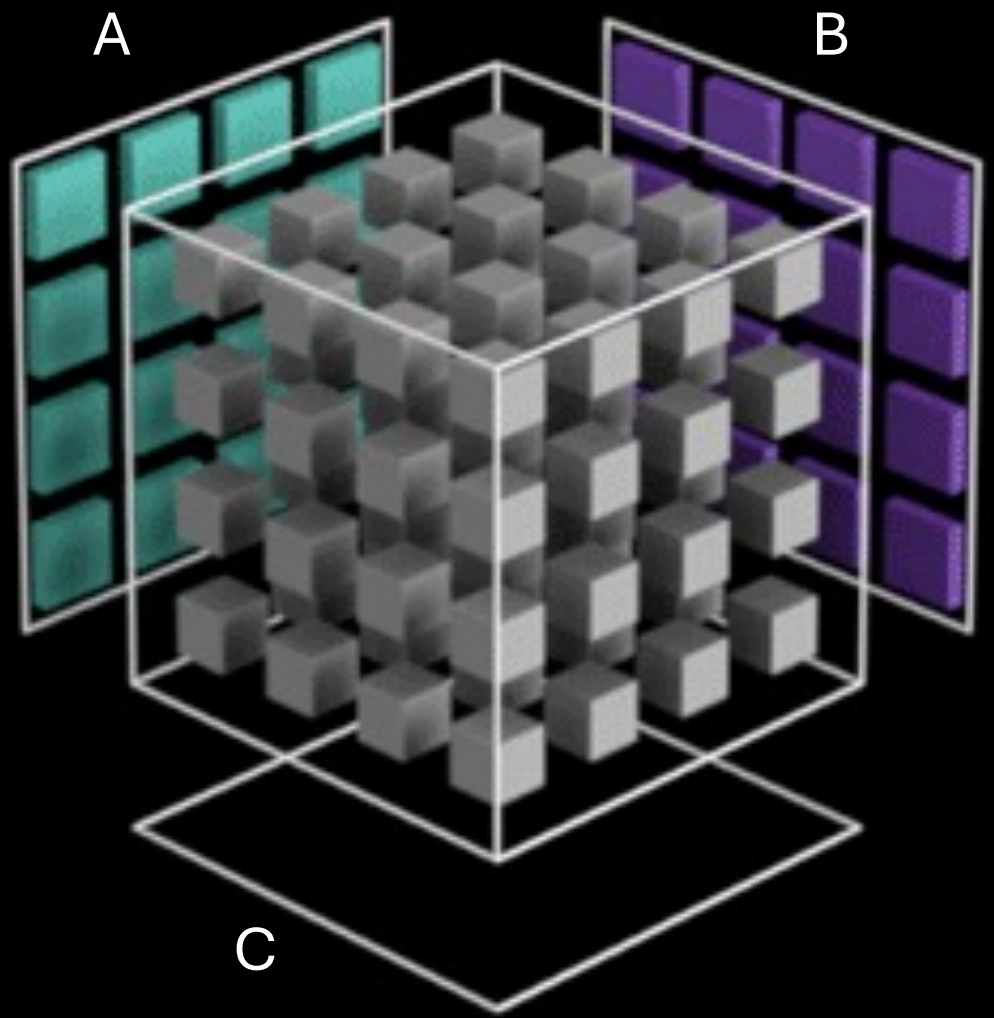


Fig. 1. C = A * B
[1]https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

Introduction to Matrix Multiplication (GEMM) Kernels

BLAS libraries decompose large GEMMs into a series of micro-kernels.

The outer product decomposition achieves (asymptotically) optimal memory data re-use

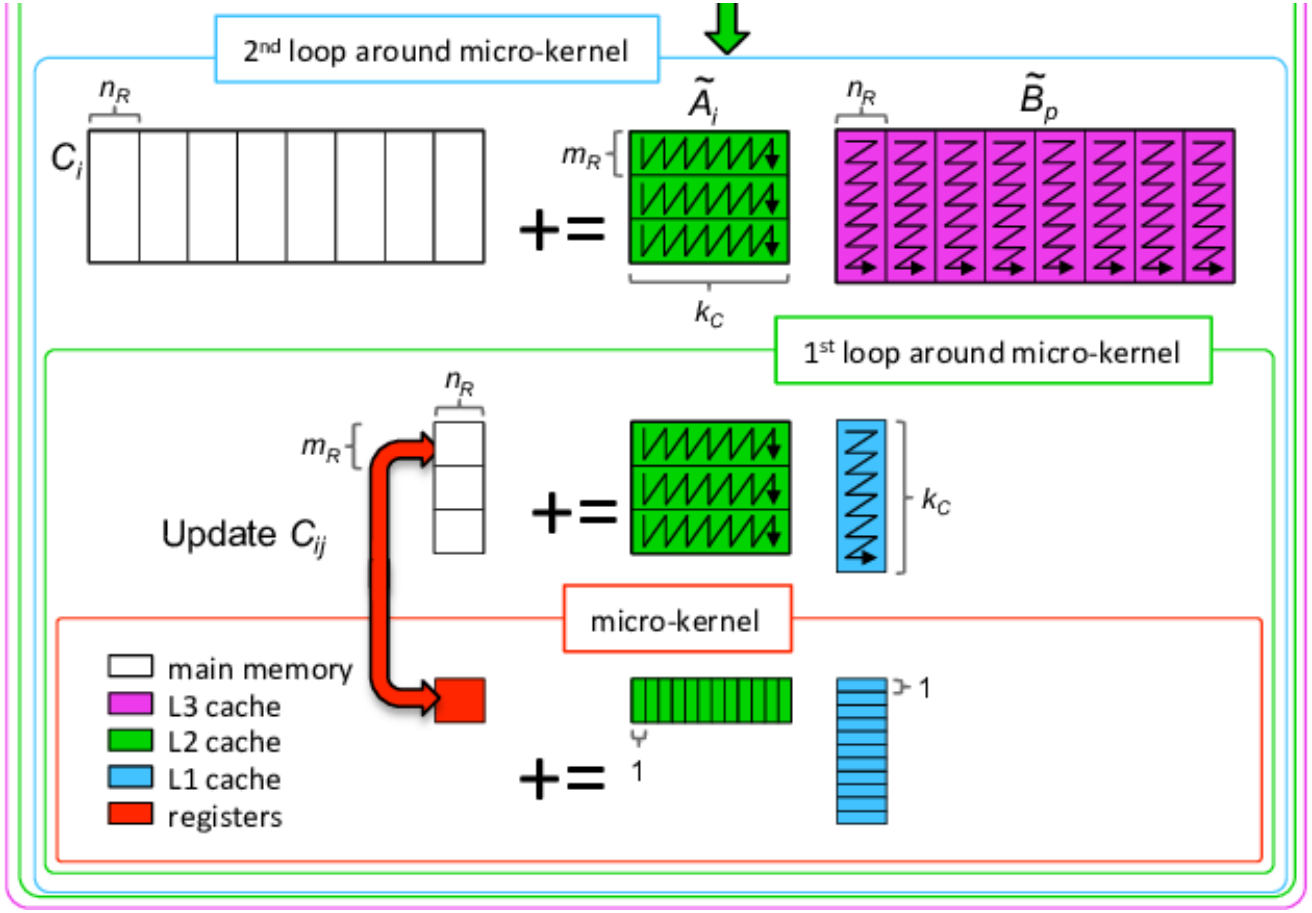


Fig 2.
[2] F. G. Van Zee and T. M. Smith, “Implementing high-performance complex matrix multiplication,” ACM Transactions on Mathematical Software, 2016

Introduction to Matrix Multiplication (GEMM) Kernels

The outer product accumulate micro-kernel (**opacc**) can be executed with just one new instruction (by grouping vector registers, eg LMUL>1)

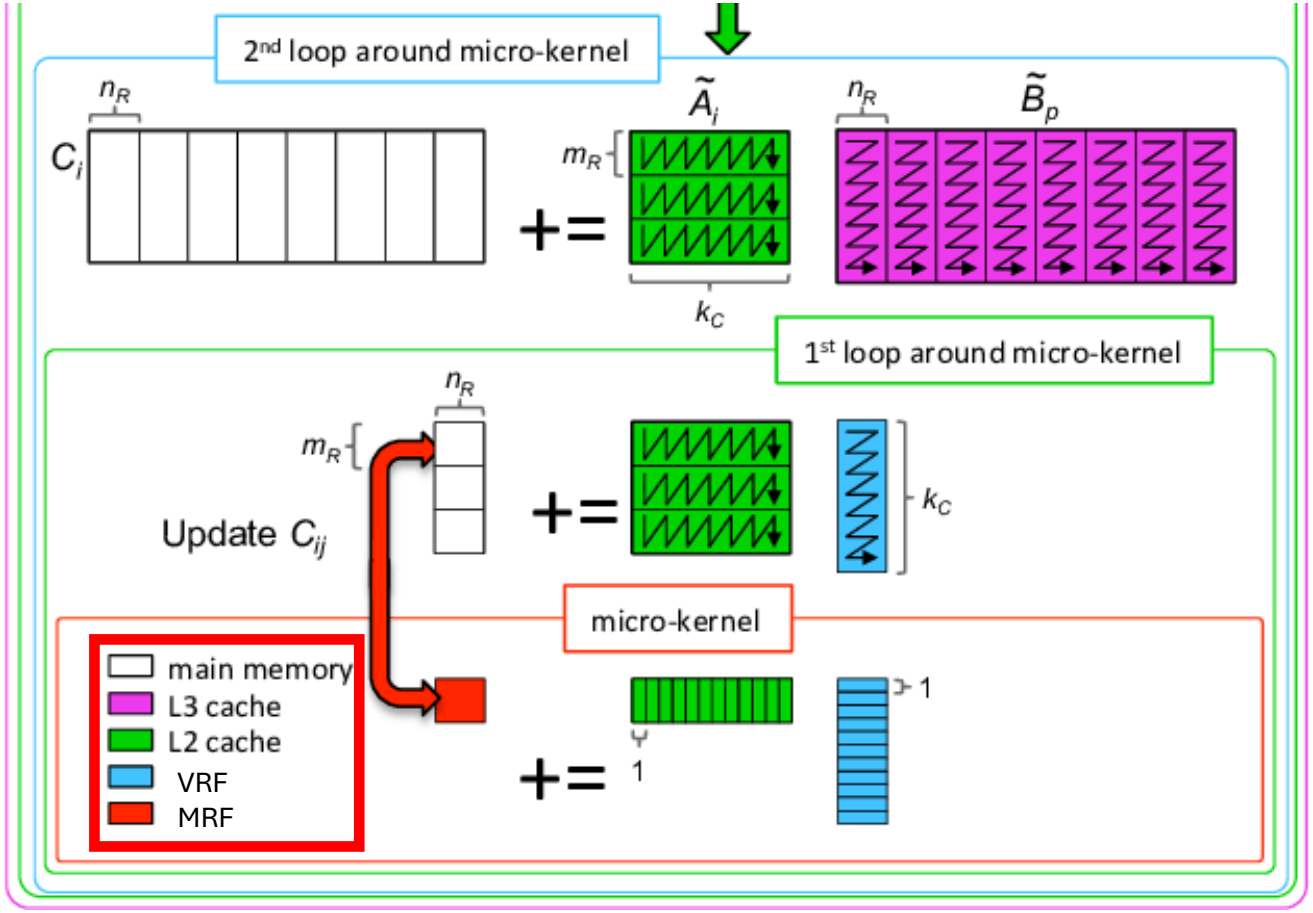


Fig 2.
[2] F. G. Van Zee and T. M. Smith, “Implementing high-performance complex matrix multiplication,” ACM Transactions on Mathematical Software, 2016

Memory Efficiency

OPACC operation:

$$m_C \mathrel{+}= v_A \otimes \mathrm{v_B}$$

For vector inputs with v_l of elements per vector, opacc performs v_l^2 MACC operations, and $2v_l$ memory operations.

Thus the operational intensity (OI) or number of MACC operations per memory operation is,

$\mathrm{OI} = \frac{v_l}{2}$

OI improves as the vector length increases

	B_0	B_1	...	B_{ml}
A_0	C_{00} += A_0 * B_0	C_{01} += A_0 * B_1	...	$C_{0,ml}$ += A_0 * B_{ml}
A_1	C_{10} += A_1 * B_0	C_{11} += A_1 * B_1	...	$C_{1,ml}$ += A_1 * B_{ml}
...	...	...	...	...
A_{vl}	$C_{vl,0}$ += A_{vl} * B_0	$C_{vl,0}$ += A_{vl} * B_0	...	$C_{vl,ml}$ += A_{vl} * B_{ml}

Memory Efficiency

OPACC operation:

$$m_C \mathrel{+}= v_A \otimes \mathrm{v_B}$$

For larger matrices, OI can be further increased by reusing data stored in fast memory

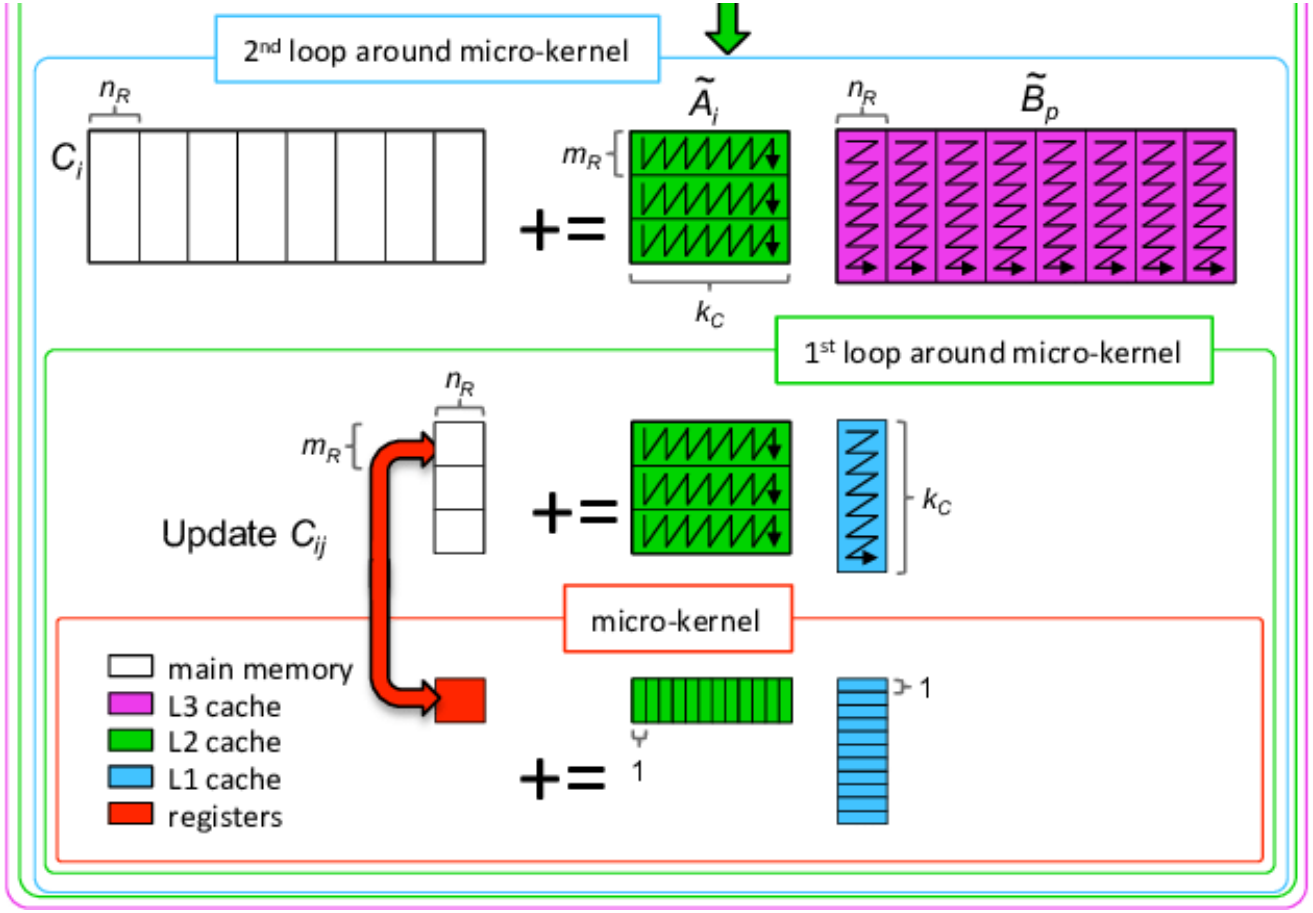


Fig 2.
[2] F. G. Van Zee and T. M. Smith, “Implementing high-performance complex matrix multiplication,” ACM Transactions on Mathematical Software, 2016

Outer Product Architecture Extension

Add matrix register state (MRF) to RVV facilitate a scalable vector length for OPACC operations

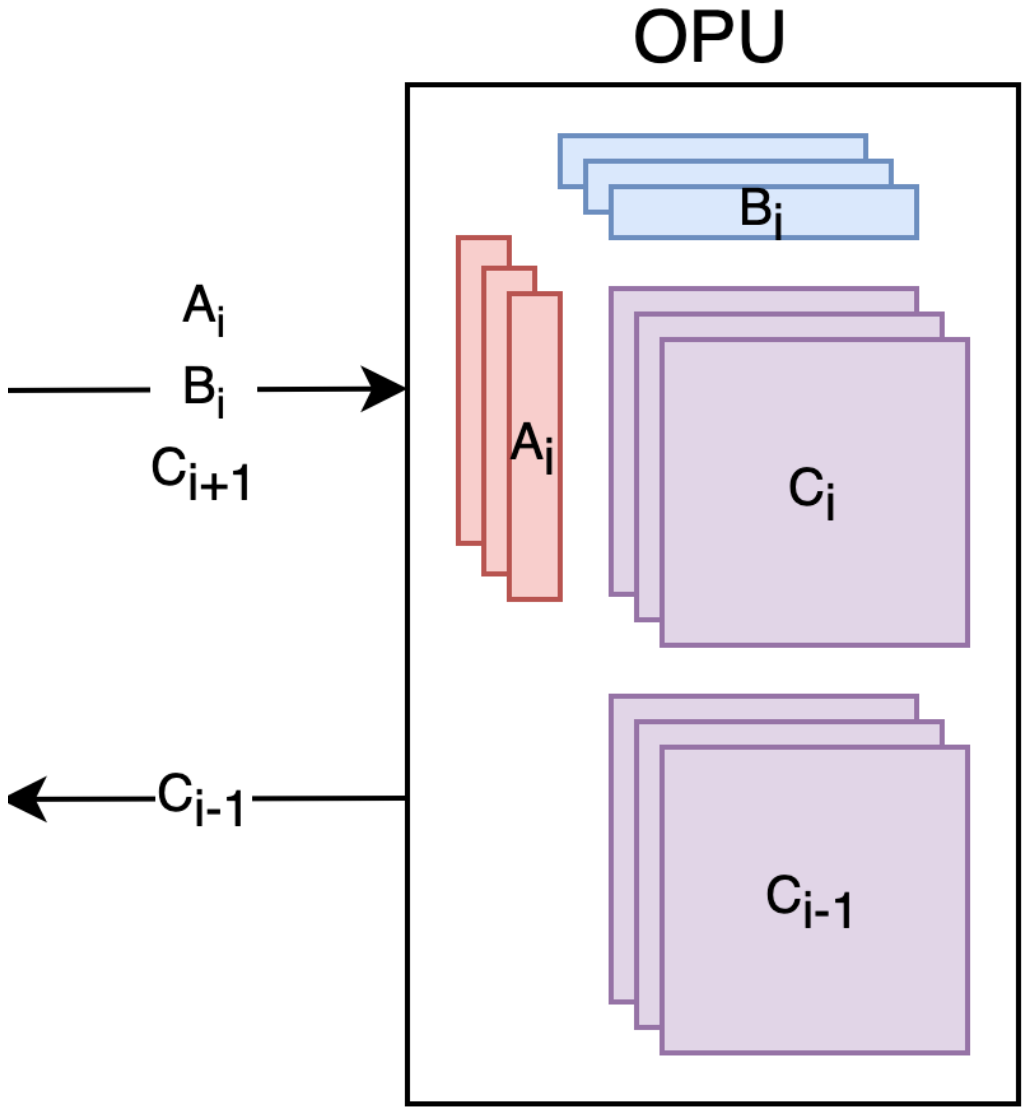
Add vmv instructions that move vectors between the VRF nad MRF:

`vmv.m.v md vs`

`vmv.v.m vd ms`

Add oppac instruction:

`opacc md vs1 vs2`

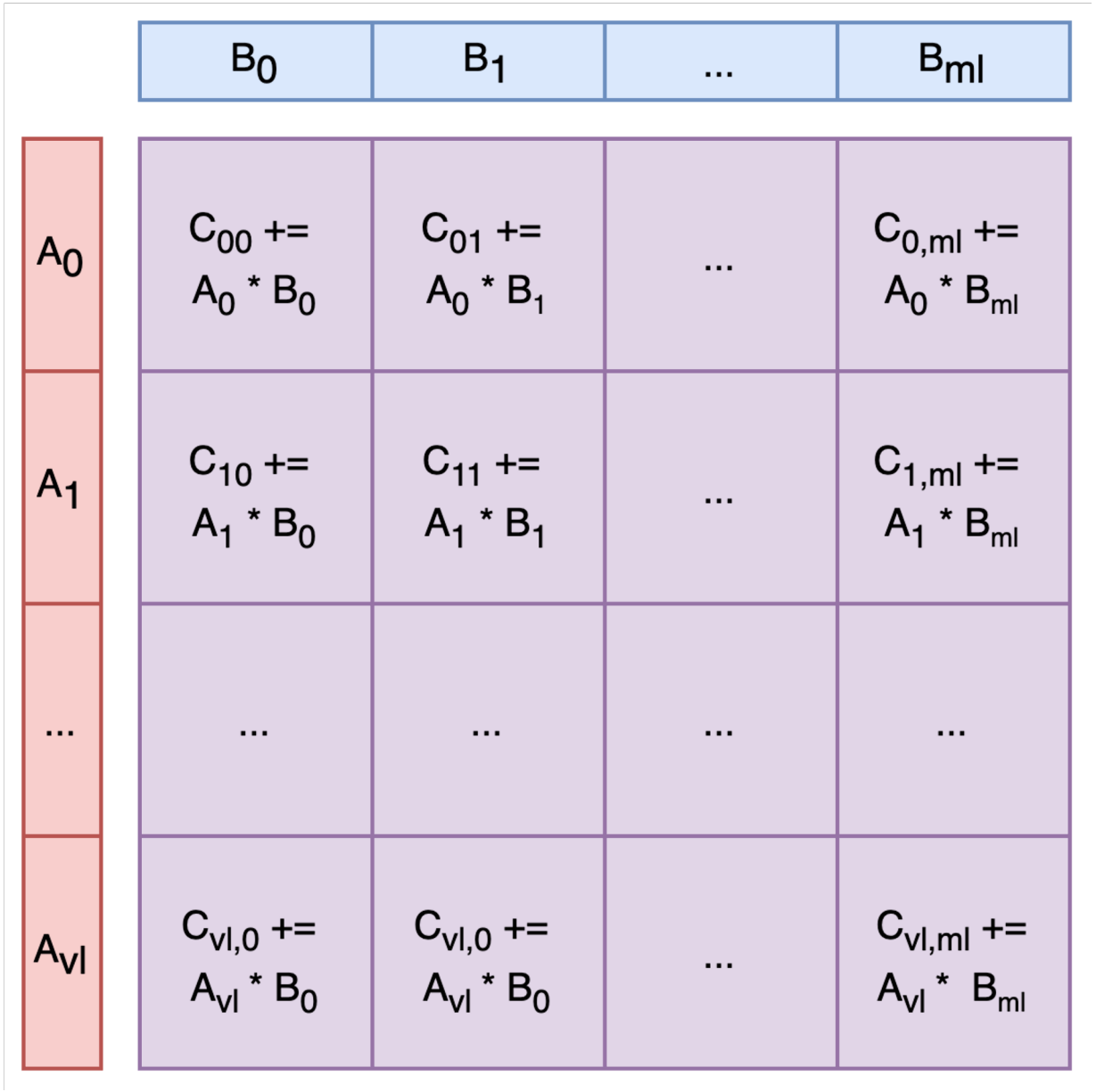


OPACC Instruction

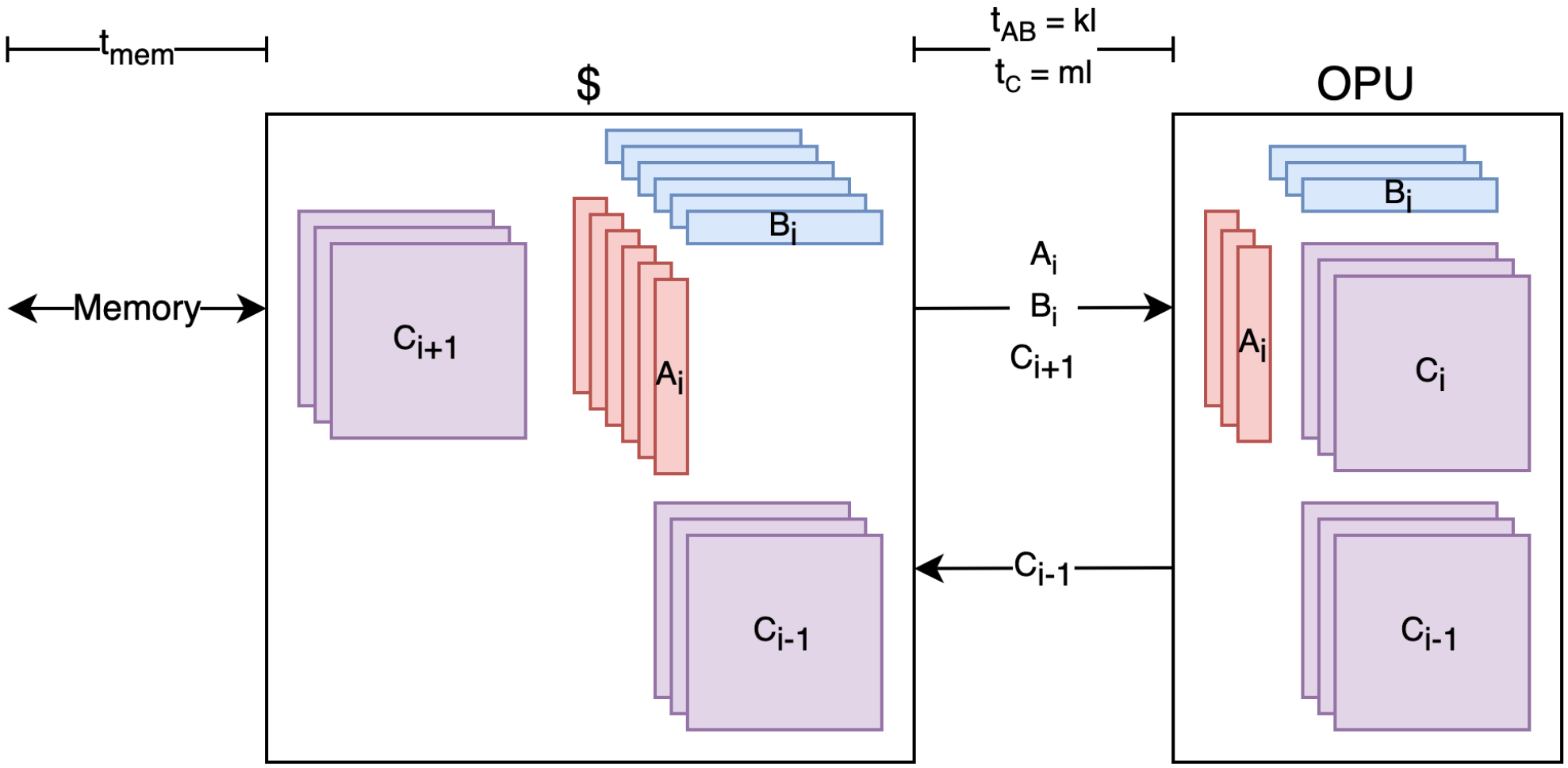
`opacc.m.v md vs1 vs2`

Performs the operation:

$$m_C \mathrel{+}= v_A \otimes \mathrm{v_B}$$

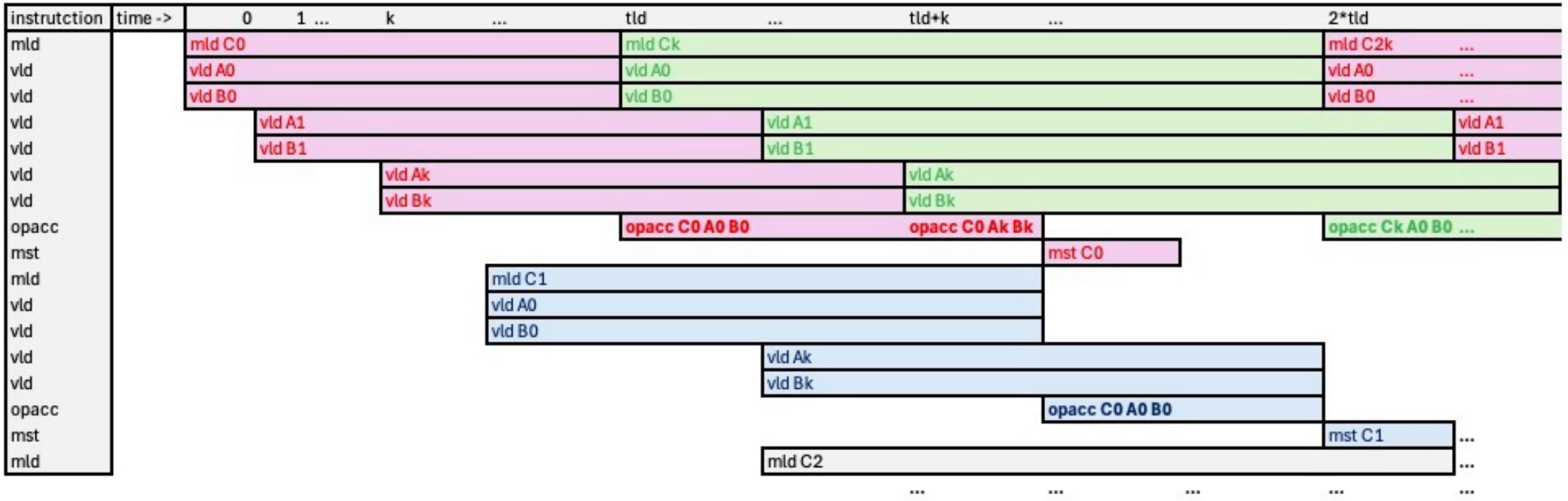


Performance Model



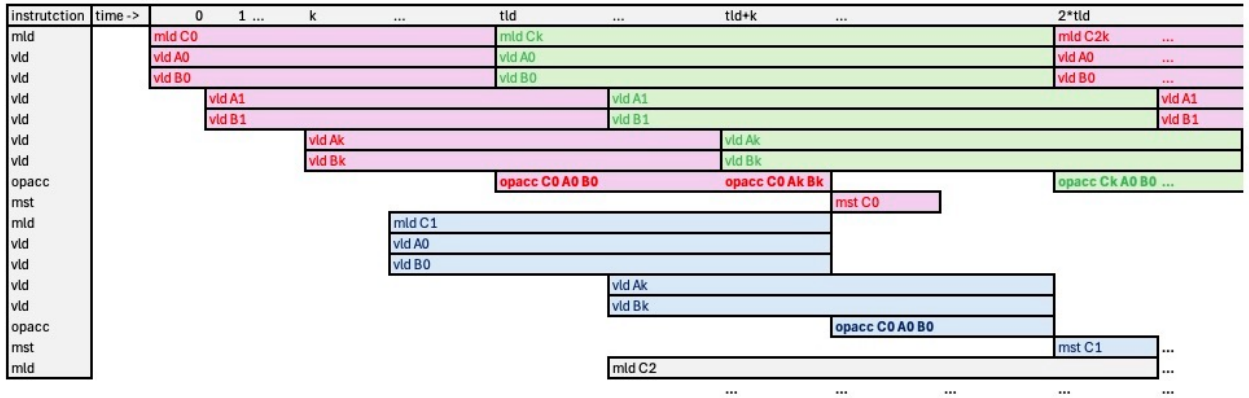
Hiding Memory Latency

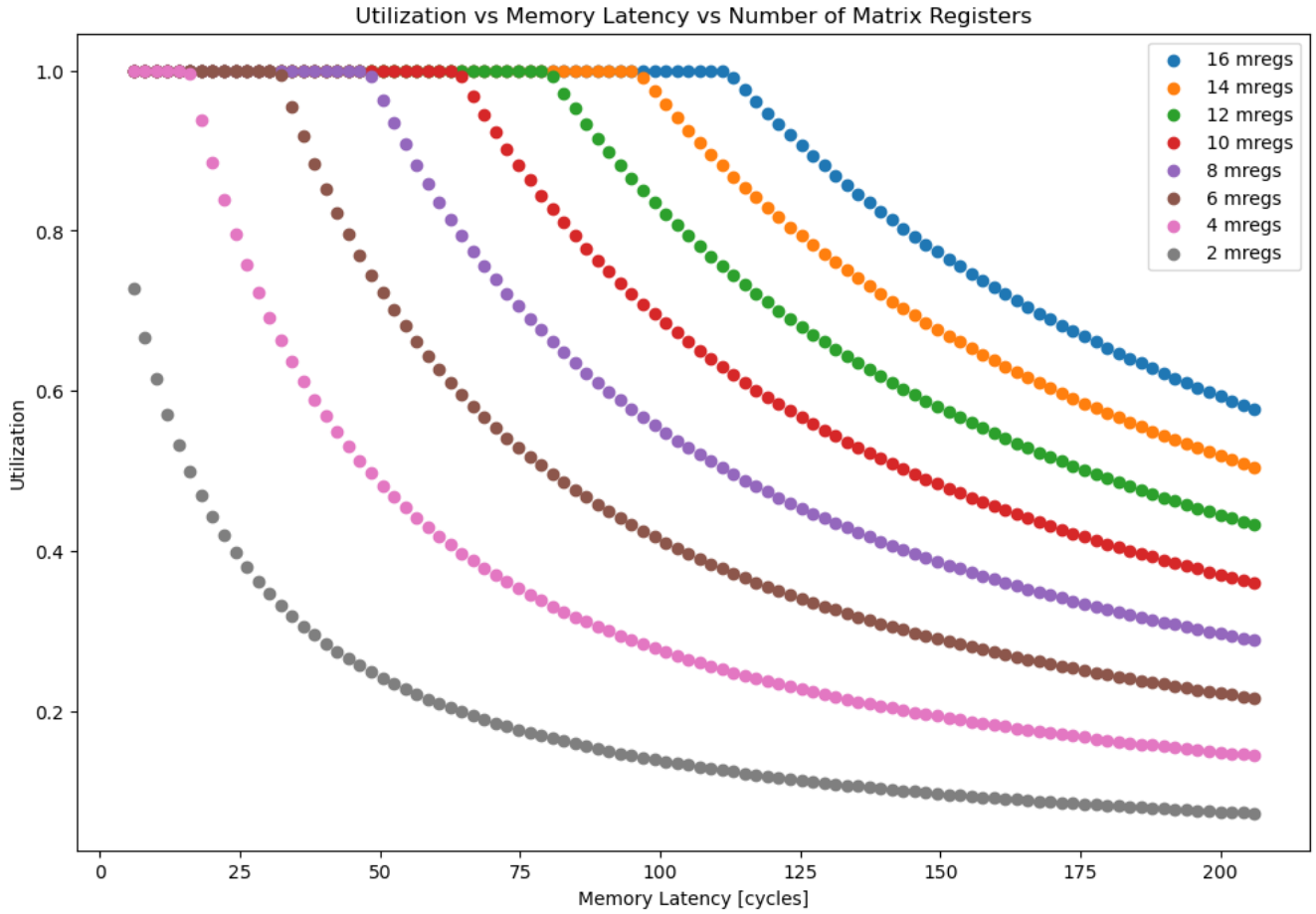
- To hide memory latency, multiple load requests must be sent to memory
- The number of requests depends on the ratio of memory latency to opacc latency



Memory Latency

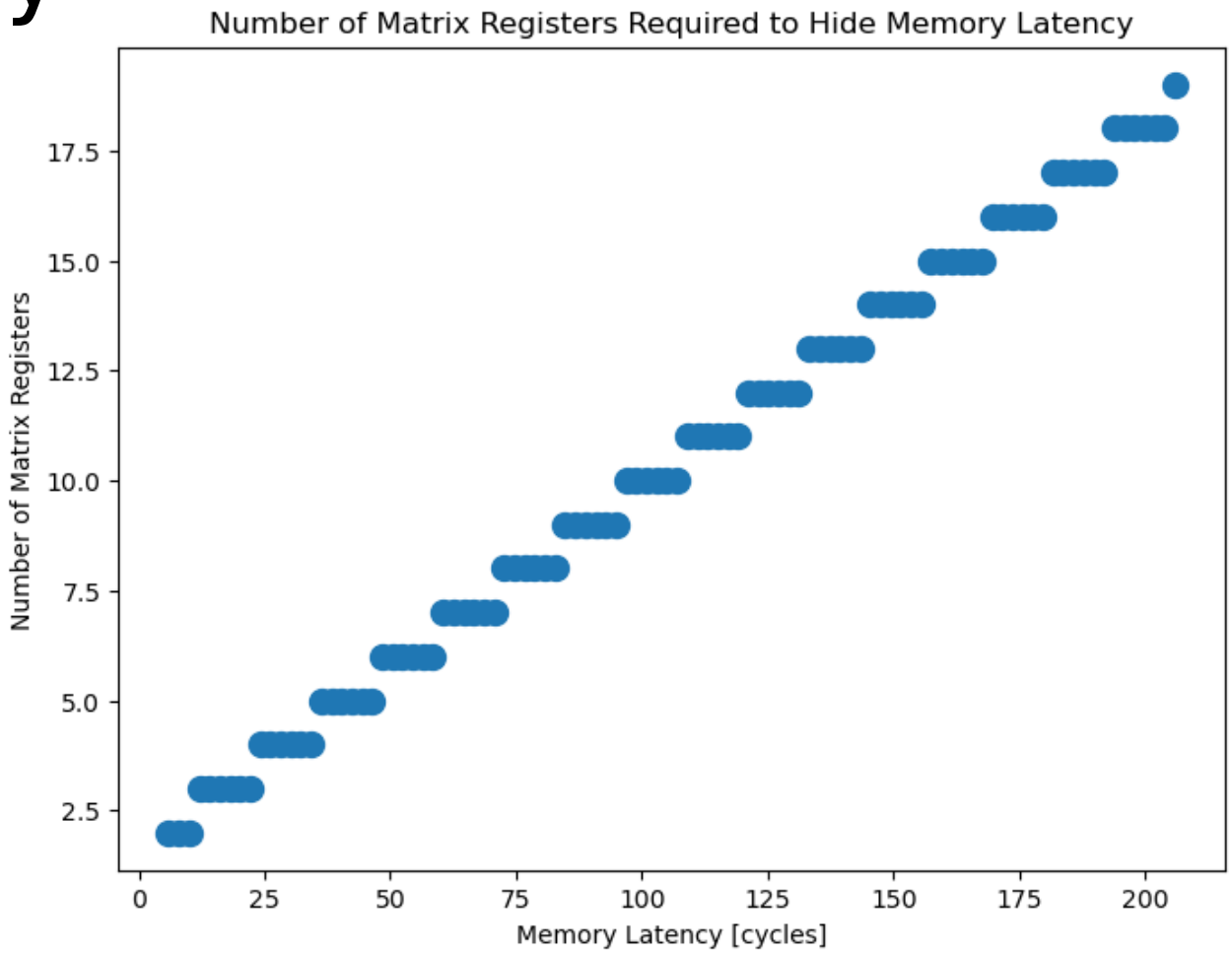
- If there aren't sufficient matrix registers to hide memory latency, the OPU utilization decreases.





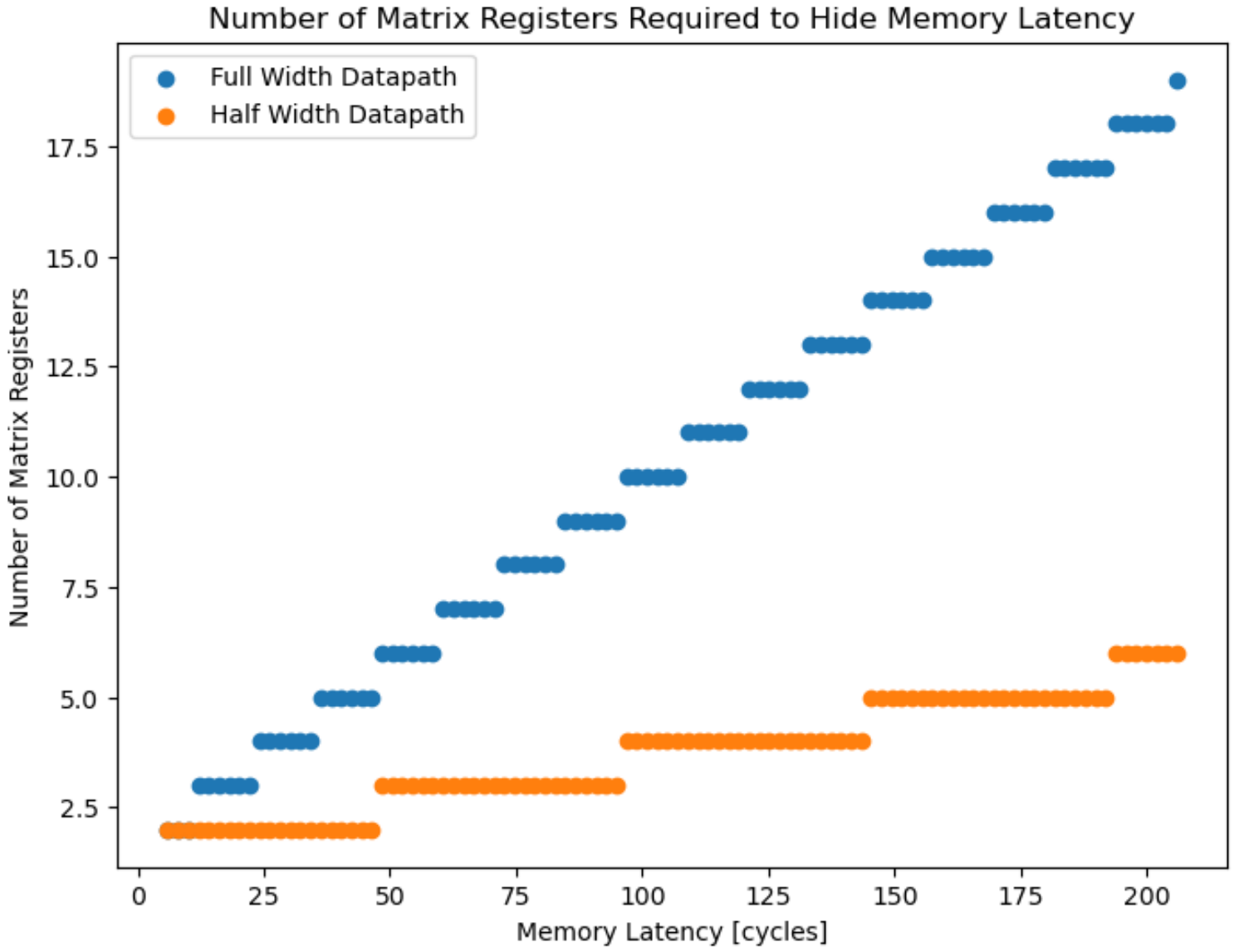
Hiding Memory Latency

- We can calculate the minimum number of registers we need to hide a given memory latency. From this we see



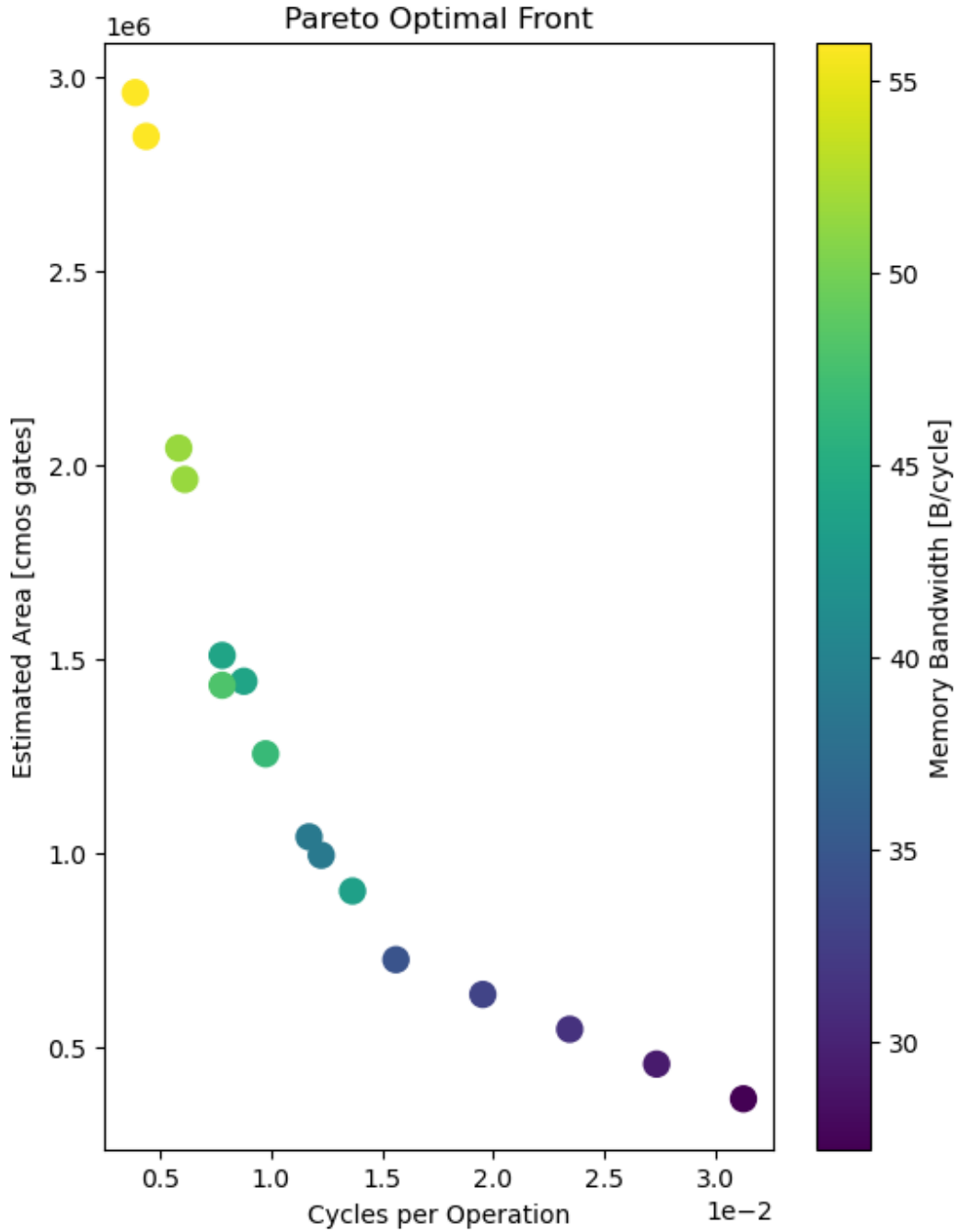
Memory Latency

- If a higher memory latency must be tolerated, the latency of the OPACC operation can be increased



Pareto-Optimal Front

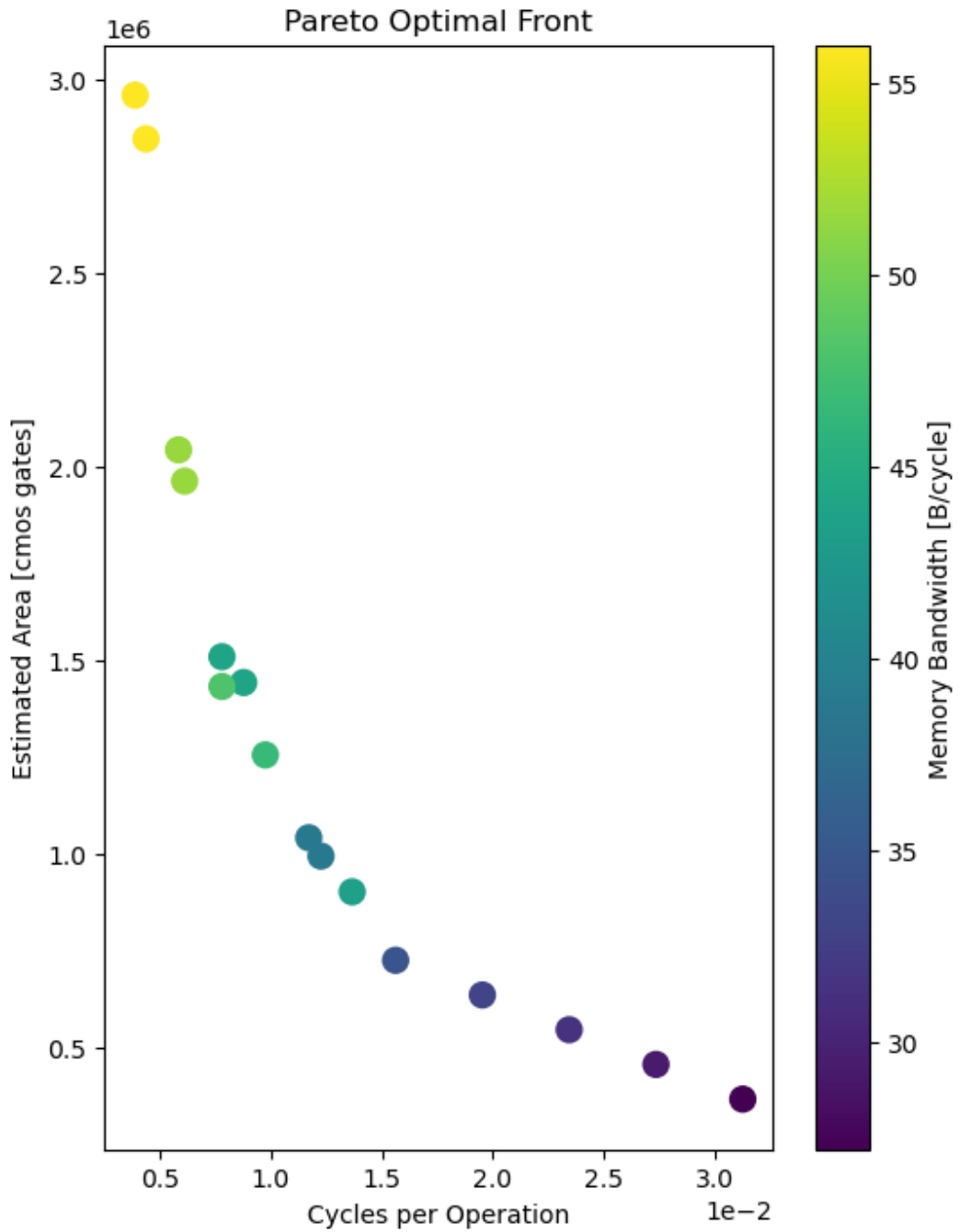
- To find optimal design points in a complex space with many tradeoffs, we find the pareto-optimal front given three performance metrics,
  - memory bandwidth,
  - logic gate count (both macc logic and registers)
  - cycles per macc operation (1/OPS)



Optimal Input Parameters

- **vlB**: vector length [Bytes]
- **mlB**: matrix length [Bytes]
- **num_mregs**: matrix register count

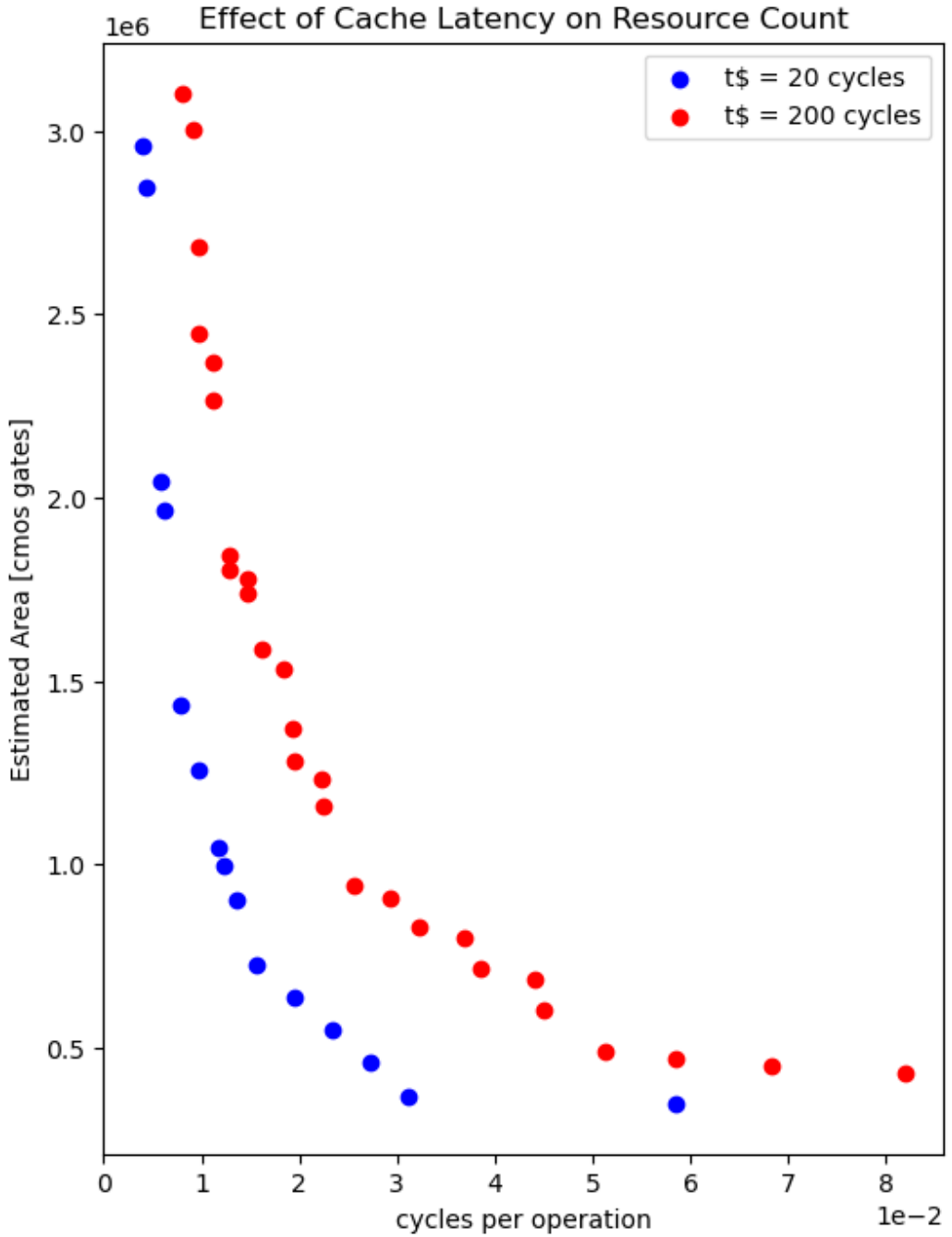
vlB	mlB	num_mregs
16.0	16.0	4
	20.0	4
	24.0	4
	28.0	4
	32.0	4
	44.0	4
		6
	64.0	4
		6
32.0	20.0	4
	28.0	4
	32.0	4
	44.0	4
		6
	64.0	4
		6



Memory Latency

- Increasing memory latency moves the pareto-front away from optimal

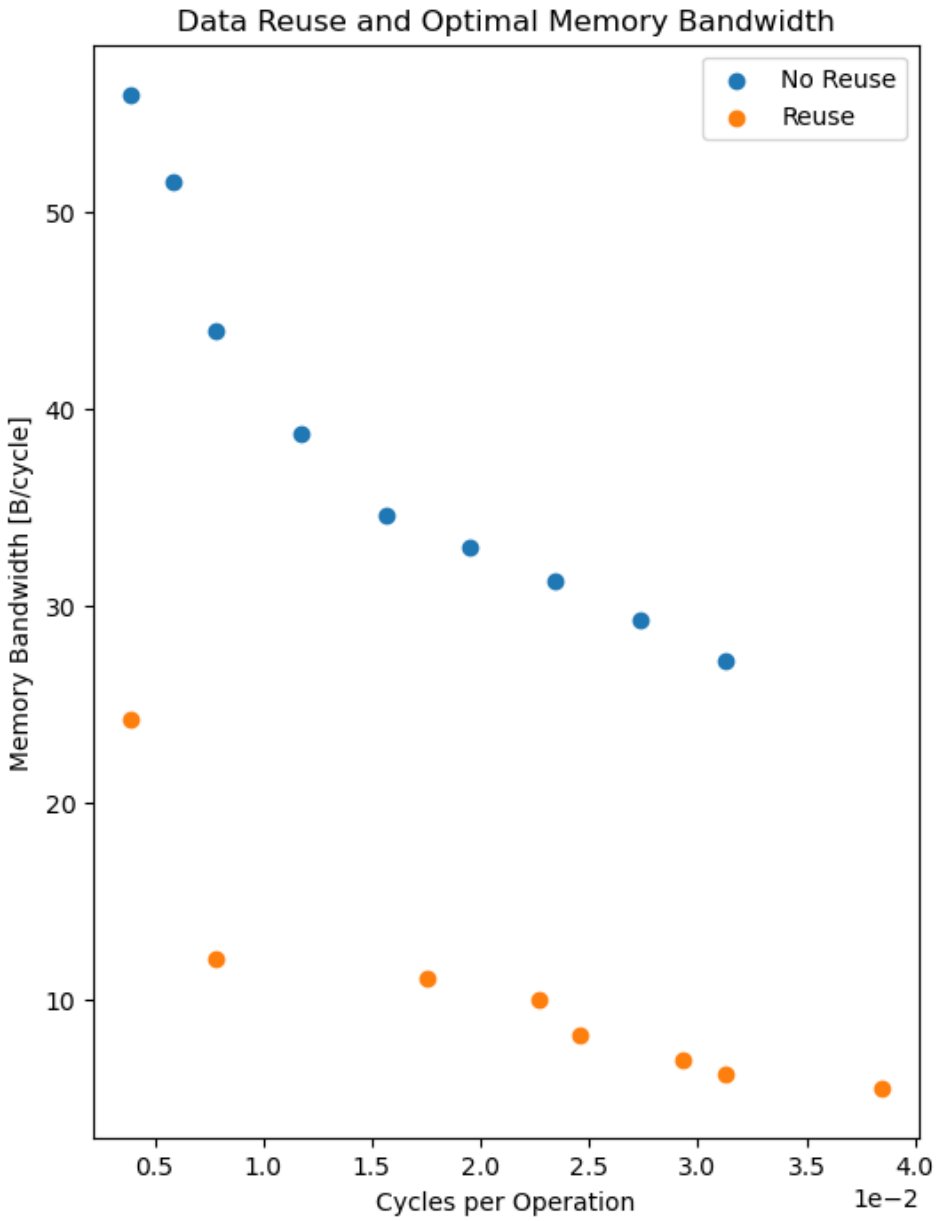
- Requires more MRF capacity, which limits performance given constrained resources



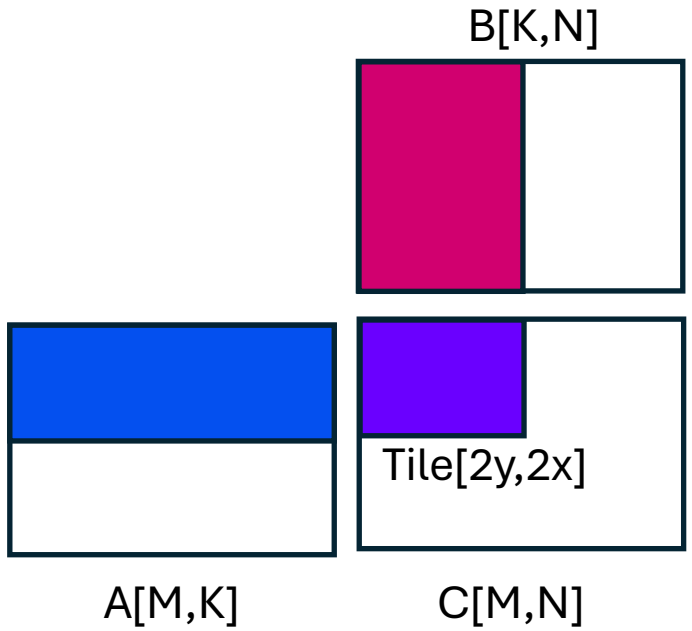
Data Reuse

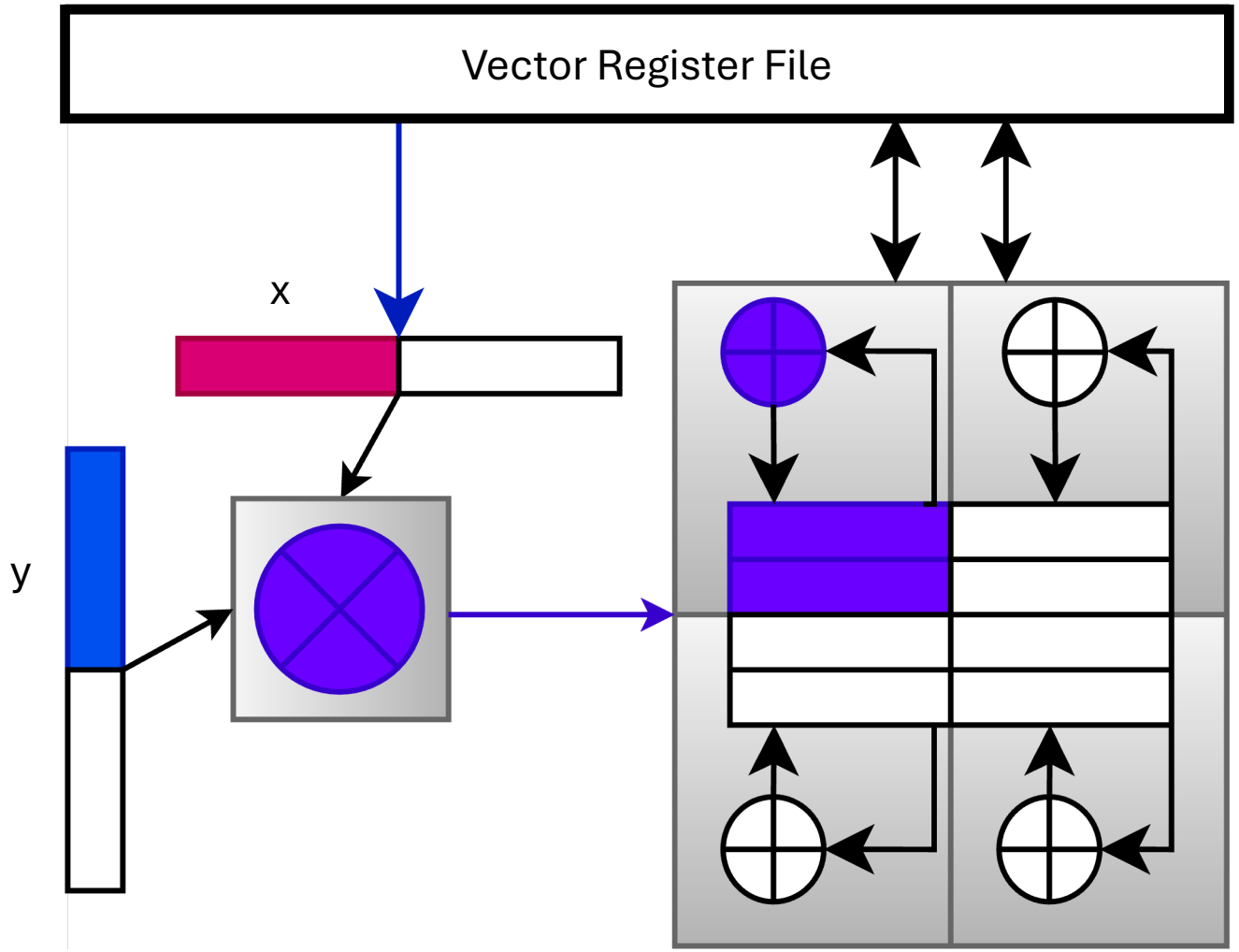
- If GEMM kernel has sufficiently large matrices, data can be reused from cache by the kernel

- We see that we can significantly reduce memory bandwidth requirements for large matrices



Half-Width Datapath

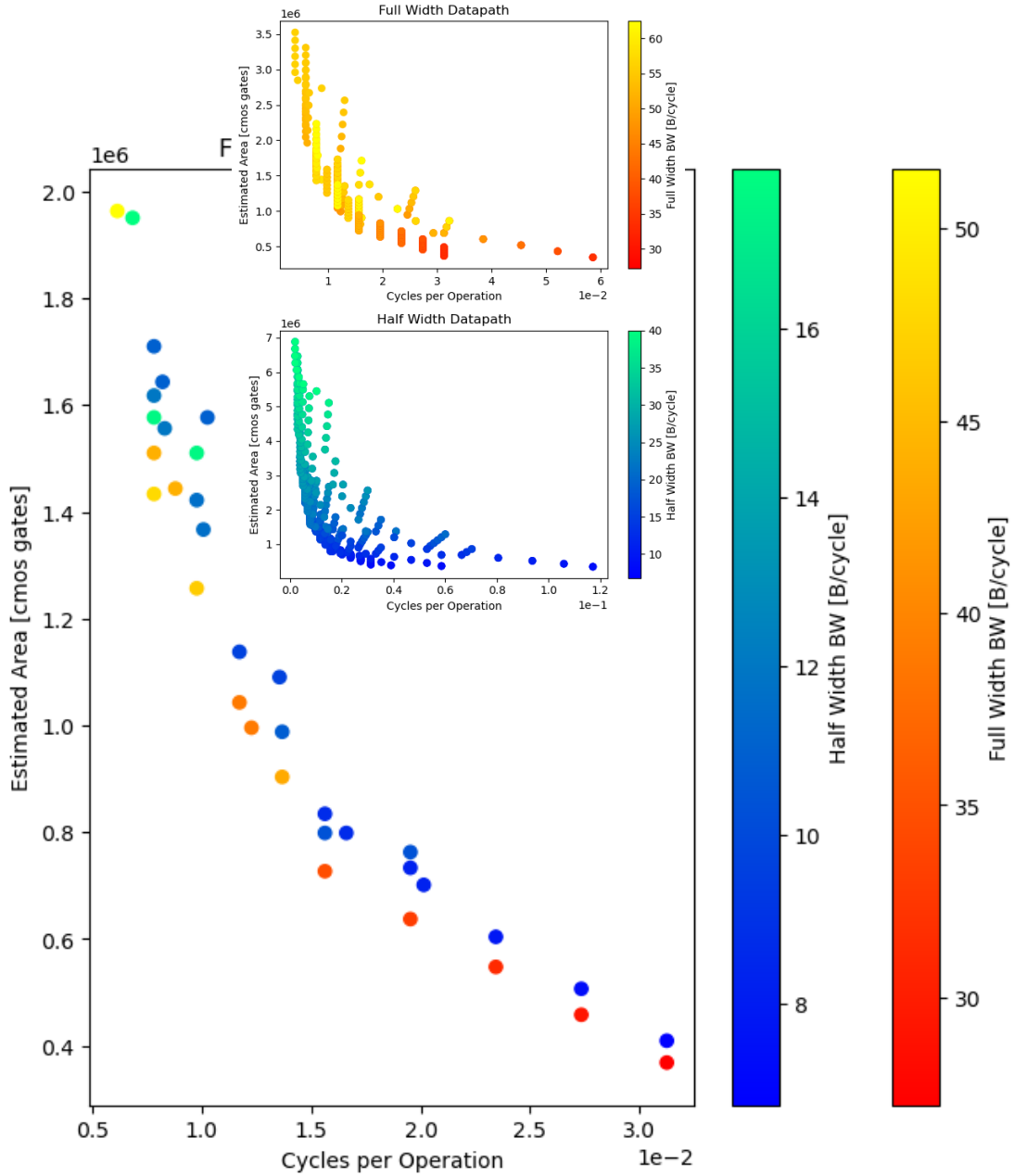




Half-Width Datapath

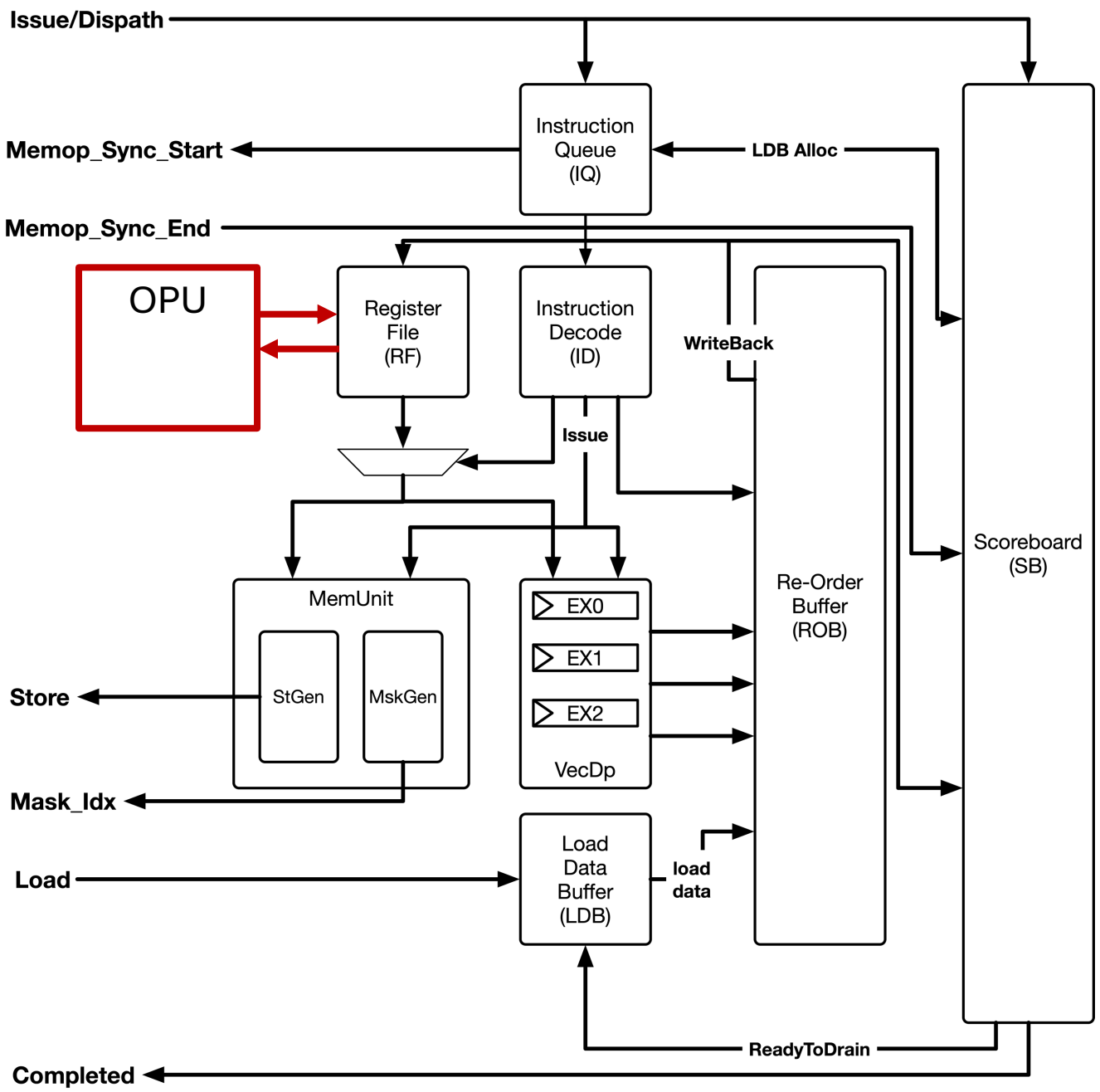
- Memory bandwidth can be reduced by reducing the width of the datapath relative to the vector length

- This trades off reduced OPS for reduced memory bandwidth



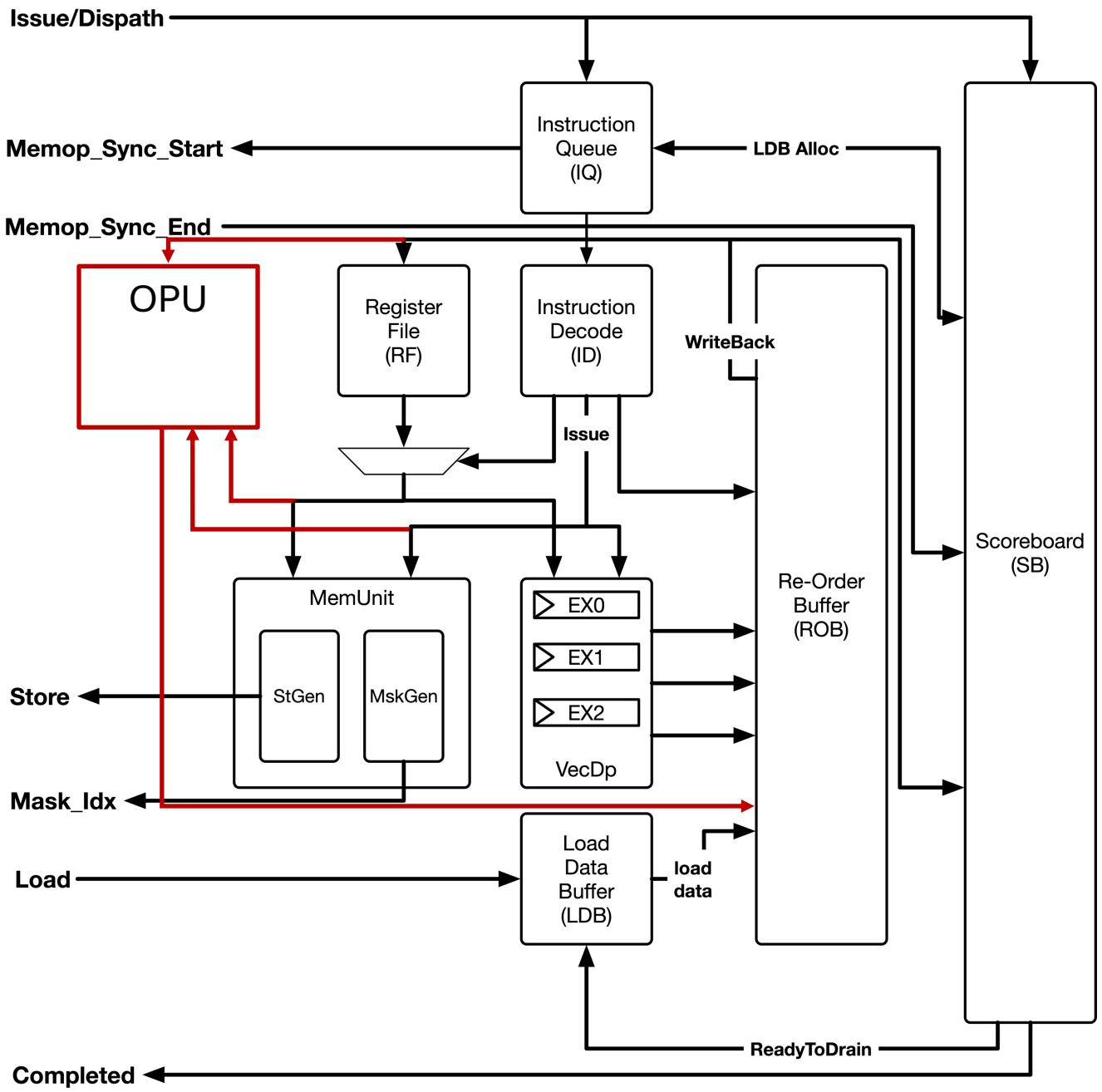
Ocelot + OPU

- Test how non-ideal micro-architecture effects performance metrics such as:
  - Functional unit utilization
  - IPC
  - Operations per cycle
  - Memory bandwidth
- Use results to test model accuracy (correlation)



Ocelot + OPU

- Test how non-ideal micro-architecture effects performance metrics such as:
  - Functional unit utilization
  - IPC
  - Operations per cycle
  - Memory bandwidth
- Use results to test model accuracy (correlation)



Example Kernel

```
//////////////////////////////////////////////
// Load 4 rows of A and B and C into VRF
//////////////////////////////////////////////
int64_t *c_temp = c;
asm volatile("vle64.v v8, (%0);" ::"r"(c_temp));c_temp += N;
asm volatile("vle64.v v0, (%0);" ::"r"(b));b += N;
asm volatile("vle64.v v4, (%0);" ::"r"(at));at += K;

… … … x 4

//////////////////////////////////////
//vmv mrf <- vrf x 4 (LMUL=1)
//////////////////////////////////////
asm volatile(".4byte 0x0004000b"); // vmv.m.v m0, v8
asm volatile(".4byte 0x0004800b");// vmv.m.v m1, v9
asm volatile(".4byte 0x0005000b");// vmv.m.v m2, v10
asm volatile(".4byte 0x0005800b");// vmv.m.v m3, v11
```

```
//////////////////////////////////////
//opacc mrf <- vrf x K=4 (LMUL=1)
//////////////////////////////////////
asm volatile(".4byte 0x0040200b");// opacc.v.m m0 v0 v4
asm volatile(".4byte 0x0050a00b");// opacc.v.m m0 v1 v5
asm volatile(".4byte 0x0061200b");// opacc.v.m m0 v2 v6
asm volatile(".4byte 0x0071a00b");// opacc.v.m m0 v3 v7


//////////////////////////////////////
//vmv vrf <- mrf x 4 (LMUL=1)
//////////////////////////////////////
asm volatile(".4byte 0x0000140b"); // mmv.v.m v8 m0
asm volatile(".4byte 0x0000948b"); // mmv.v.m v9 m1
asm volatile(".4byte 0x0001150b"); // mmv.v.m v10 m2
asm volatile(".4byte 0x0001958b"); // mmv.v.m v11 m3

//////////////////////////////////////
// store results
//////////////////////////////////////
c_temp = c;
asm volatile("vse64.v v8, (%0);" ::"r"(c_temp)); c_temp += N;
asm volatile("vse64.v v9, (%0);" ::"r"(c_temp)); c_temp += N;
asm volatile("vse64.v v10, (%0);" ::"r"(c_temp));c_temp += N;
asm volatile("vse64.v v11, (%0);" ::"r"(c_temp));
```

Results

(4 x 4) x (4 x 4) matrix multiplication:

VPU execution took **447 cycles**.
VPU performance is **286 OPs/1000 cycles**.

OPU execution took **241 cycles.**
OPU performance is **531 OPs/1000 cycles**.

Speedup = 1.85
Ideal speedup is $v_l = 4$

Speed up should improve with larger K

Results

Small vs Mega Bobcat (Boom) Frontend

Small Bobcat
execution took **376 cycles.**
performance is **340 OPs/1000 cycles.**

speedup suggests OPU may be limited by instruction decode bandwidth

Mega Bobcat
execution took **241 cycles.**
performance is **531 OPs/1000 cycles**.

Correlation Results

- Given the parameters:
  - VLEN = MLEN = 256
  - 64-bit datatype
  - M=K=N=4
- Ideal Model Ops/1000cycles = 39,000 (vs ~300)
- The microarchitecture overhead is significant for this small GEMM

Future Work

- Open Source Ocelot + OPU (?)
  - Add support for all RISC-V datatypes
  - Add scalable matrix length MLEN
  - Test performance on the full BLAS decomposition

- Get more performance and correlation data: (utilization, IPC, memory bandwidth)
- Show model correlation improves for larger matrix dimensions

- Run synthesis and place and route to correlate modelled area estimates with physical design

- Tune performance of OPU GEMM kernels