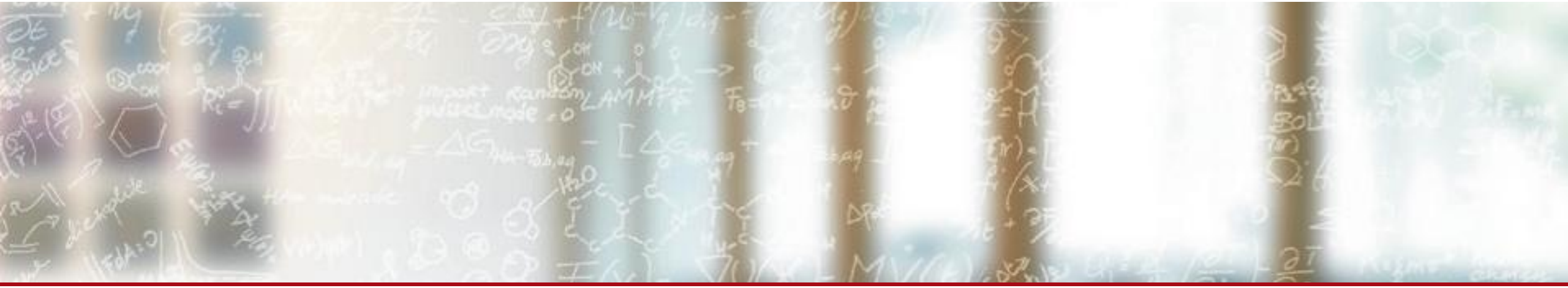
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Performance Portable Thread Cooperation on AMD and NVIDIA GPUs

Sebastian Keller
May 10th, 2023

Performance tuning for the AMD CDNA-2 architecture

- Achieving the advertised performance on AMD Mi250X can be quite challenging

- Architectural differences between NVIDIA and AMD chips:
  - Register file
  - Shared memory

- Do these differences affect software implementation choices for optimal performance?

AMD CDNA-2 specifics vs. NVIDIA

- CDNA has a 512 KB (vector) register file per compute unit (CU), for values that differ between wave (warp) lanes
- Plus a 12 KB scalar register files for values identical among threads in a wave
- Local data store (shared memory) on separate area chip

- NVIDIA Ampere: 256 KB register file
- Shared memory resides on register file

References: AMD CDNA, AMD CDNA2, NVIDIA Ampere

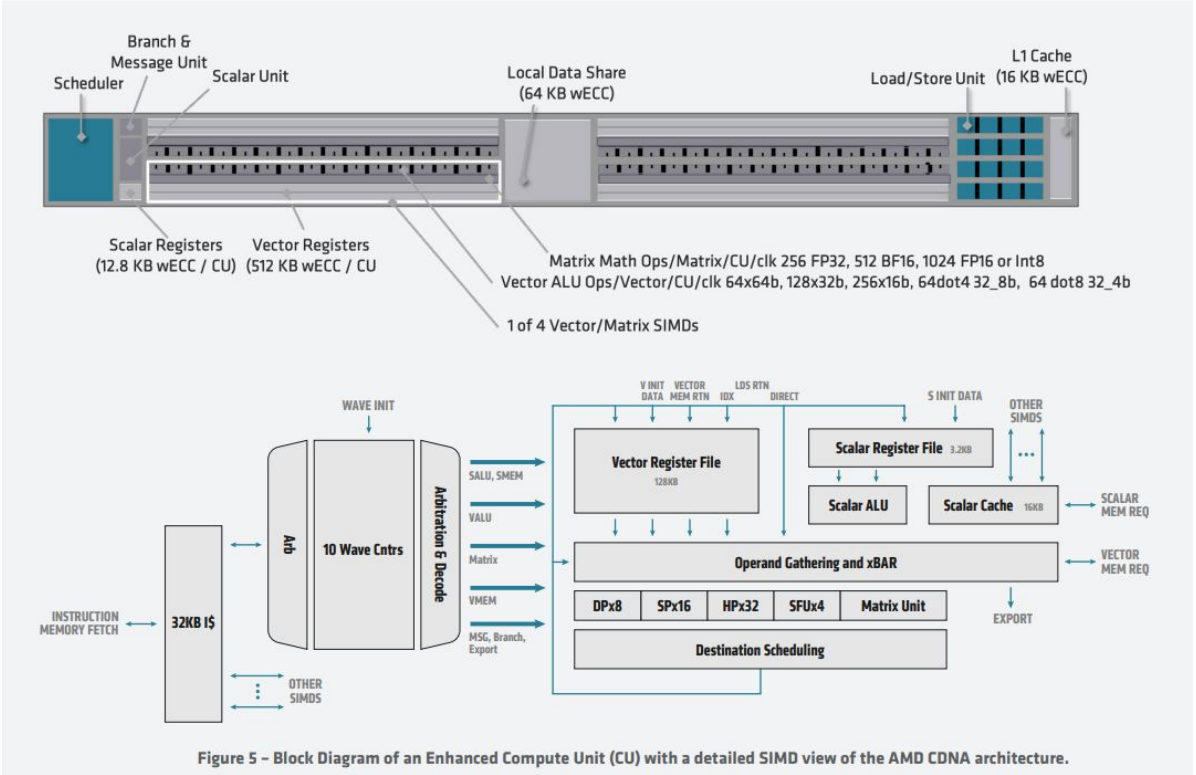
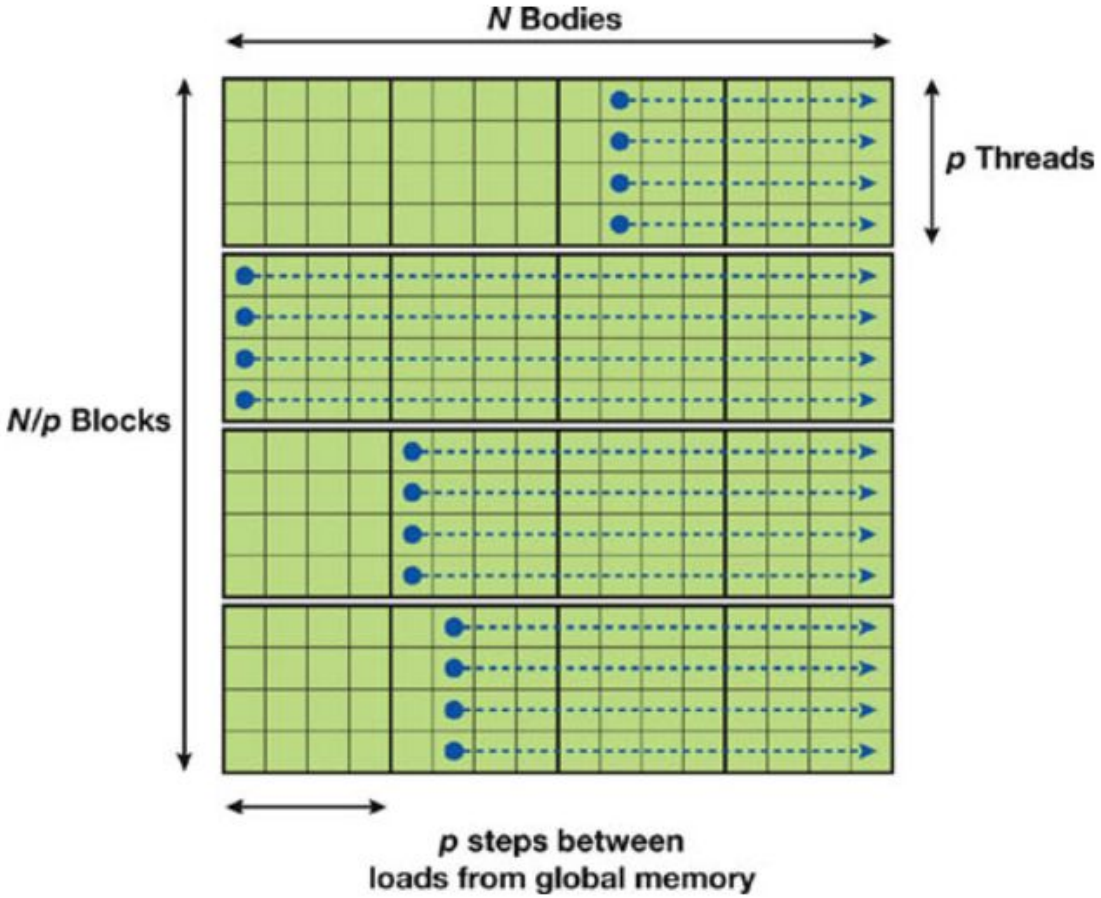


Figure 5 – Block Diagram of an Enhanced Compute Unit (CU) with a detailed SIMD view of the AMD CDNA architecture.

Classic N-body problem on AMD and NVIDIA cards

- Part of the Fast Multipole Method (FMM) and treecodes for electrostatics and gravity
- Compute N^2 interactions between N bodies
  Each square is a particle-particle interaction pair
- Two options to cache loads from global memory: shared memory or intra-warp exchanges
- Each (4x4) tile is a group of cooperating threads,
  - thread block with shared mem
  - warp/wave of 32 or 64 threads
- An NxN tile computes N^2 interaction pairs with N instead of N^2 loads

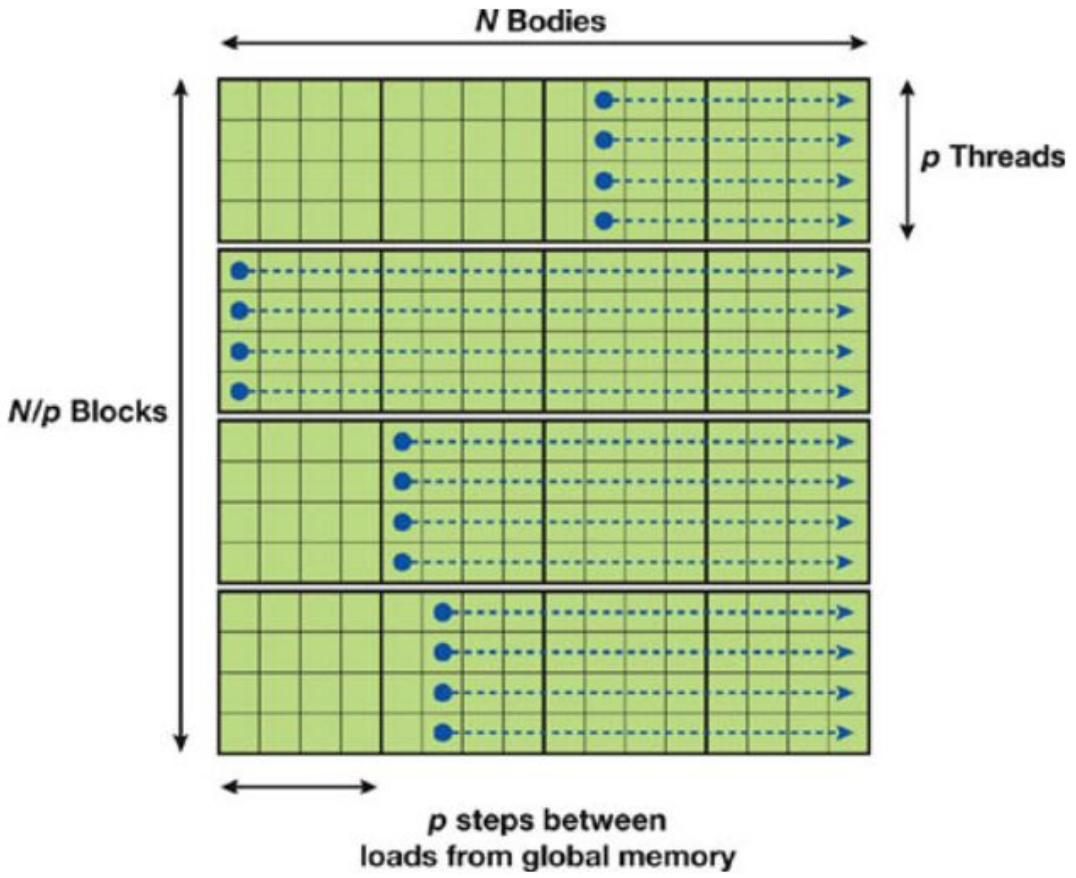


Ref: GPU Gems 3

Classic N-body and AMD and NVIDIA cards

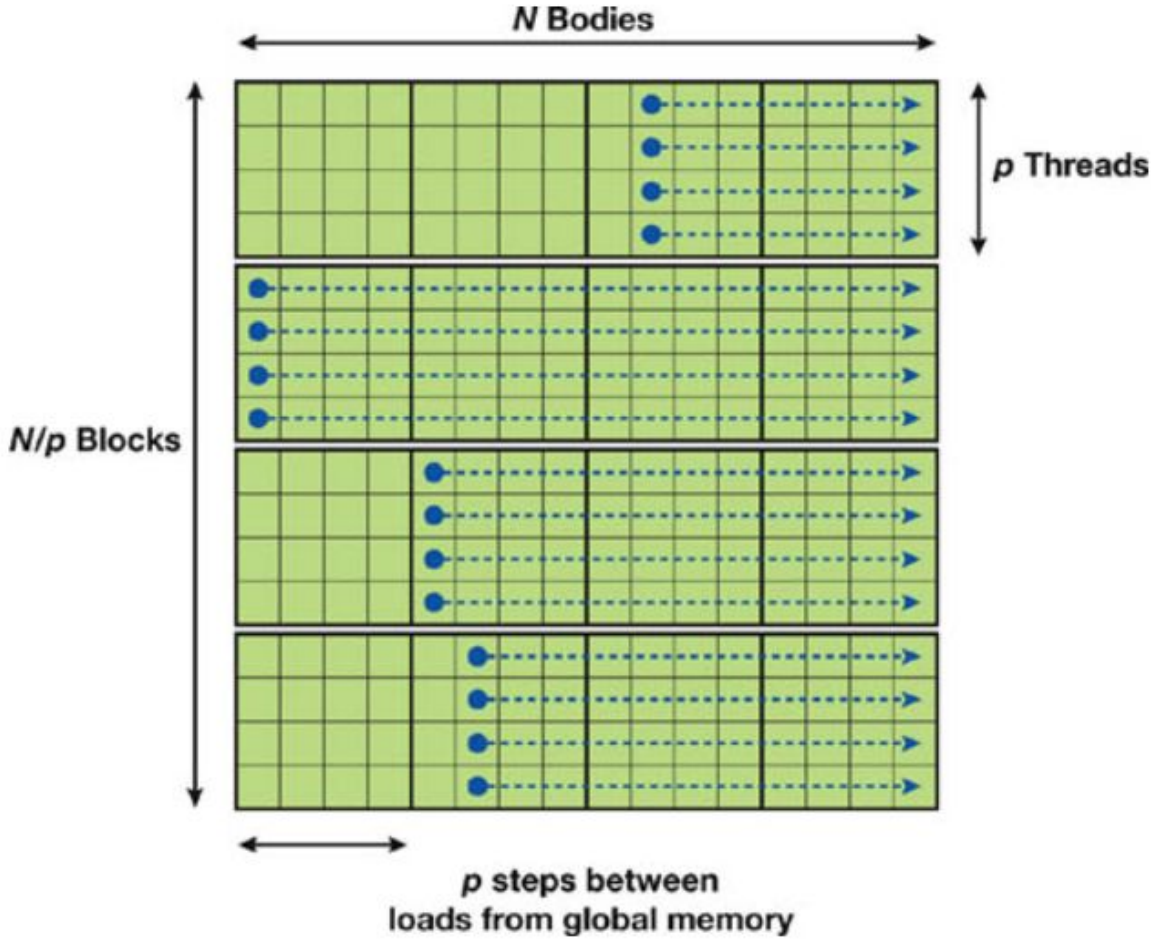
- A single interaction between two particles

```
template<class T> __device__
Vec4<T> p2p(Vec4<T> acc, const Vec3<Tc>& pos_i,
            const Vec3<Tc>& pos_j, T m_j)
{
    Vec3<Tc> dX = pos_j - pos_i;        // 3 flops
    Tc       R2 = norm2(dX) + eps2;     // 6 flops

    Tc invR   = rsqrt(R2);              // 2 flops (sqrt + div)
    Tc invR2  = invR * invR;            // 1 flop
    Tc invR3m = m_j * invR * invR2;     // 3 flops

    acc[0] -= invR3m * R2;              // 2 flops
    acc[1] += dX[0] * invR3m;           // 2 flops
    acc[2] += dX[1] * invR3m;           // 2 flops
    acc[3] += dX[2] * invR3m;           // 2 flops

    return acc;                         // 23 flops
}
```



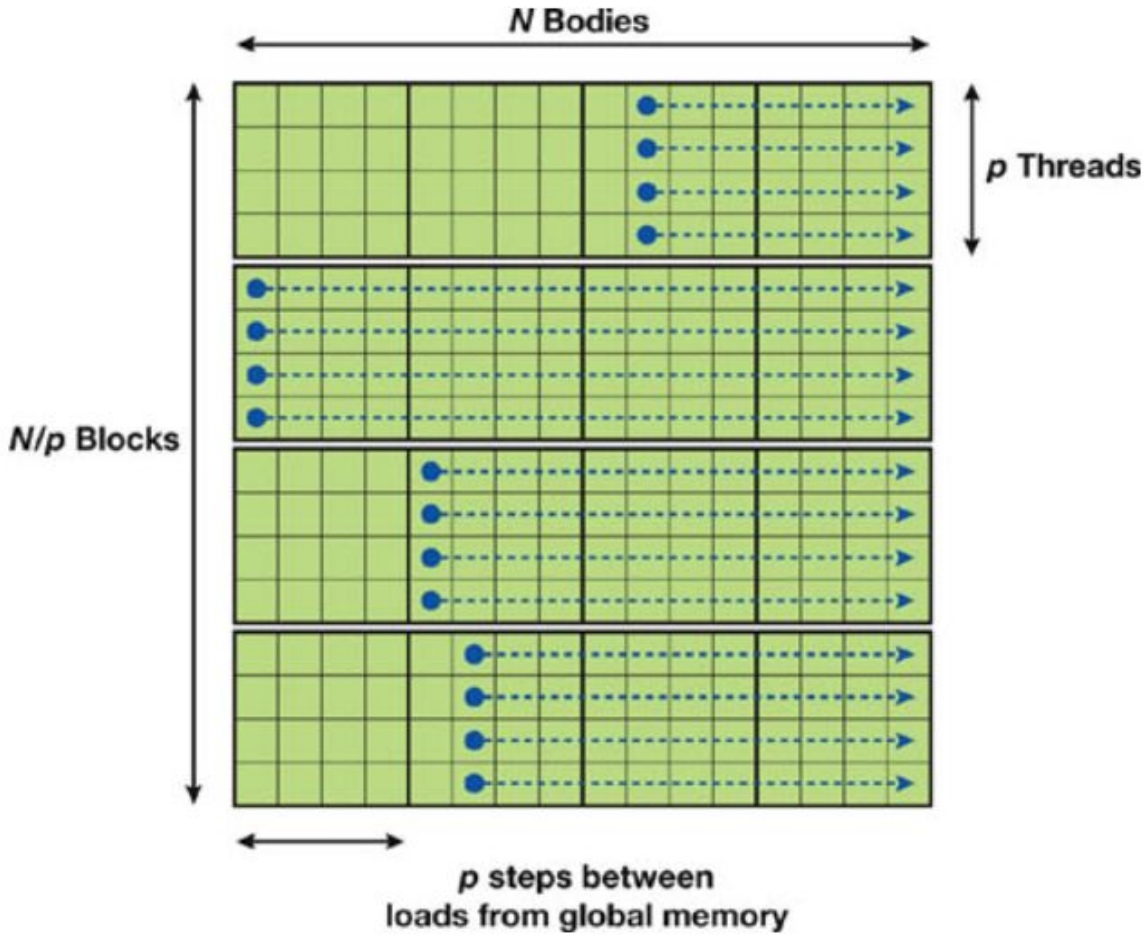
First option: shared memory

```
template<class T>
__global__ void directShared(T* x, T* y, T* z, T* m,
                             T* ax, T* ay, T* az)
{
    unsigned targetIdx = blockDim.x * blockIdx.x + threadIdx.x;

    Vec3<T> pos_i x[targetIdx], y[targetIdx], z[targetIdx]};
    Vec3<T> acc = 0;
    __shared__ util::array<T, 4> sm_body[blockDim.x];

    // loop over sources
    for (unsigned tile = 0; tile < gridDim.x; ++tile)
    {
        int s = tile * blockDim.x + threadIdx.x;
        sm_bodytile[threadIdx.x] = {x[s], y[s], z[s], m[s]};
        __syncthreads();

        // reuse values in shared mem
        for (int j = 0; j < blockDim.x; ++j)
        {
            Vec3<T> pos_j{sm_bodytile[j][0], sm_};
            T m_j = sm_bodytile[j][3];

            acc = p2p(acc, pos_i, pos_j, m_j);
        }
        __syncthreads();
    }

   // store target in ax, ay, az
}
```



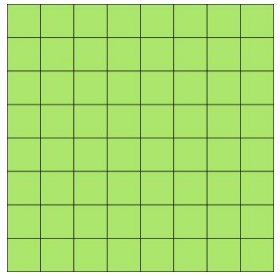
Second option: intra-warp exchange with shuffles

```
template<class T>
__global__ void directShfl(T* x, T* y, T* z, T* m,
                           T* ax, T* ay, T* az)
{
    unsigned targetIdx = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned laneIdx   = threadIdx.x & (warpSize - 1);

    Vec3<T> pos_i x[targetIdx], y[targetIdx], z[targetIdx]};
    Vec3<T> acc = 0;

    // loop over sources
    for (unsigned s = laneIdx; s < N; s += warpSize)
    {
        util::array<T, 4> sourceBody{x[s], y[s], z[s], m[s]};

        // reuse values in warp
        for (int l = 0; l < warpSize; ++l)
        {
            Vec3<T> pos_j{shfl(sourceBody[0], l),
                          shfl(sourceBody[1], l),
                          shfl(sourceBody[2], l)};
            T m_j = shfl(sourceBody[3], l);

            acc = p2p(acc, pos_i, pos_j, m_j);
        }
    }

   // store target in ax, ay, az
}
```

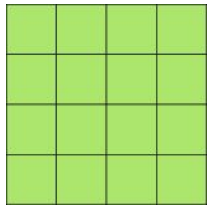


Comparison between shared-mem and warp-shuffle approach

Shared-mem



warp-shuffles



✅ Bigger tile size = higher flop/byte

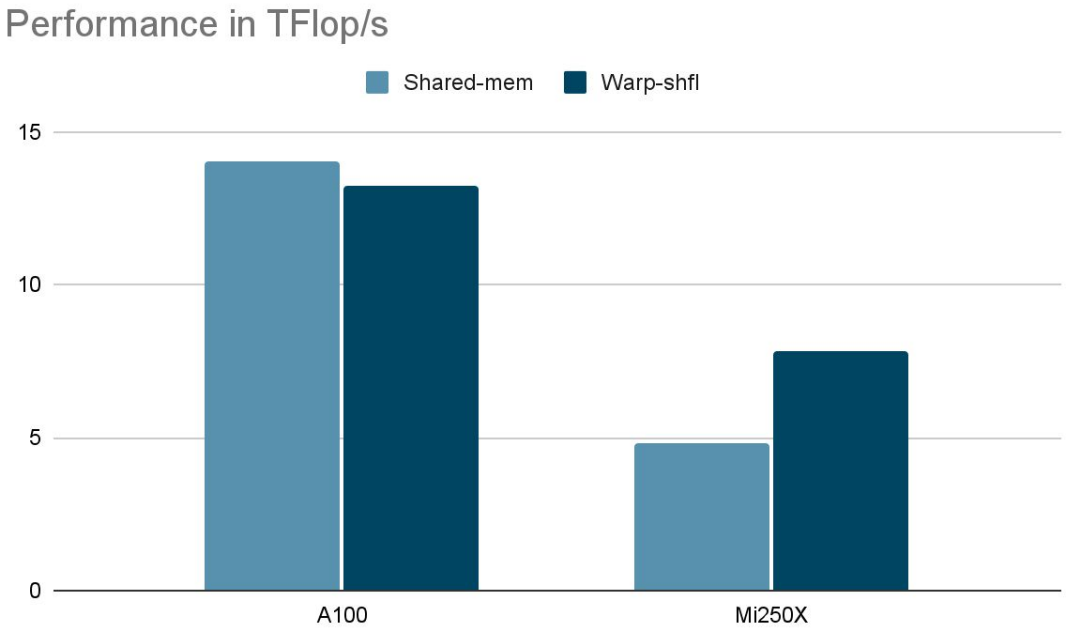
❌ SM synchronization overhead

❌ Tile size limited by vector unit length
= less flop/byte

✅ No synchronization for warp-shuffles

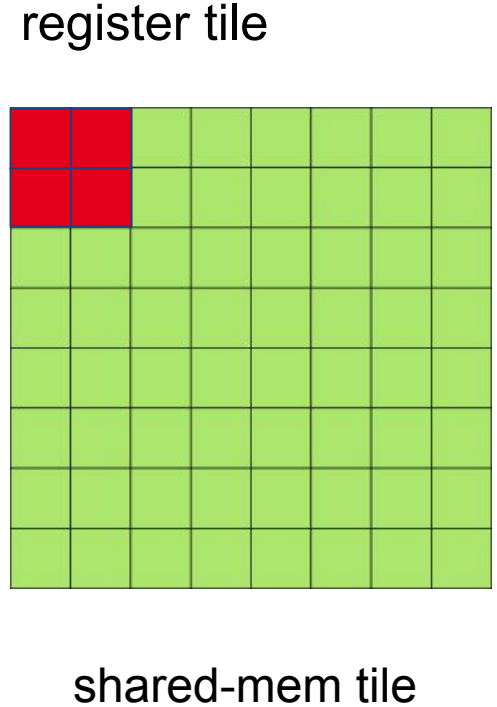
Comparison between shared-mem and warp-shuffle approach

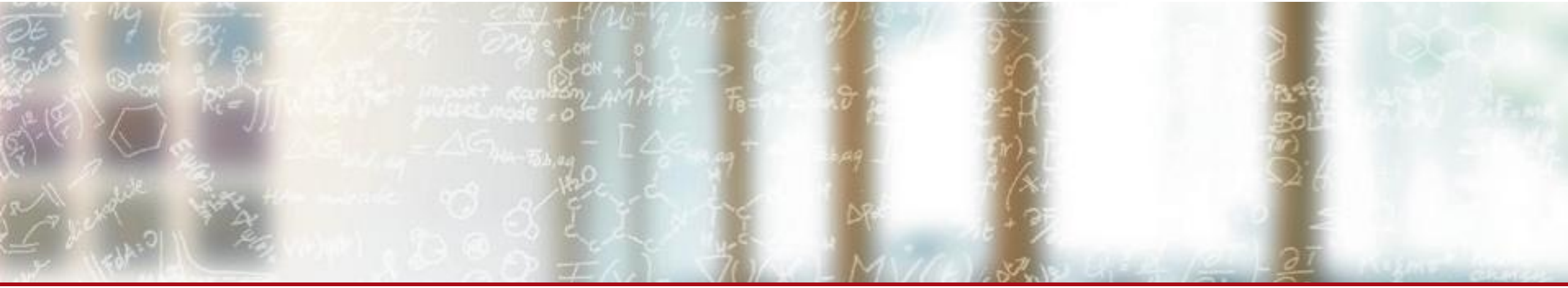
- Faster shared memory and smaller warp size (32) favors shared-memory version on NVIDIA Ampere
- Slower shared memory and larger wave size (64) translates to warp-shuffles outperforming the shared memory approach on AMD CDNA2



Combination of both approaches: shared-mem and register tiles

- With tiling for both shared-mem and registers both devices reach ~95% performance of the best version
- Usual tricks such as `__restrict__` -ed pointers applied
- `rsqrt` instruction not a bottleneck
- All blocking strategies are highly compute bound on both devices.





Summary

- Intra-warp data exchanges between threads preferable to shared-mem on AMD MI250X
- Combination of warp and shared-mem approaches is the best compromise that’s portable
- AMD GPUs are a little trickier to optimize for