Divergence-Aware Warp Scheduling

Programmability

Simple, But Divergent

```
__global__ void
spmv_csr_scalar_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float* out)
{
    int myRow = blockIdx.x * blockDim.x
        + threadIdx.x;
    texReader vecTexReader;

    if (myRow < dim)
    {
        float t = 0.0f;
        int start = rowDelimiters[myRow];
        int end = rowDelimiters[myRow+1];
        // Divergent Branch
        for (int j = start; j < end; j++)
        {
            // Uncoalesced Loads
            int col = cols[j];
            t += val[j] * vecTexReader(col);

        }
        out[myRow] = t;
    }
}
```

Divergence

- Divergent loads a major problem when programming GPUs
  - Common in irregular applications
- Programmer encouraged to restructure program, make use of scratchpad
  - Our work asks: What if they didn't have to?

Hardware aware of code locality can take advantage of it without needing the programmer

Less Divergence, But More Complicated

```
__global__ void
spmv_csr_vector_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float * out)
{
    int t = threadIdx.x;
    int id = t & (warpSize-1);
    int warpsPerBlock = blockDim.x / warpSize;
    int myRow = (blockIdx.x * warpsPerBlock)
            + (t / warpSize);
    texReader vecTexReader;

    __shared__ volatile
        float partialSums[BLOCK_SIZE];

    if (myRow < dim)
    {
        int warpStart = rowDelimiters[myRow];
        int warpEnd = rowDelimiters[myRow+1];
        float mySum = 0;
        for (int j = warpStart + id;
             j < warpEnd; j += warpSize)
        {
            int col = cols[j];
            mySum += val[j] * vecTexReader(col);
        }
        partialSums[t] = mySum;

        // Reduce partial sums
        if (id < 16)
            partialSums[t] += partialSums[t+16];
        if (id <  8)
            partialSums[t] += partialSums[t+ 8];
        if (id <  4)
            partialSums[t] += partialSums[t+ 4];
        if (id <  2)
            partialSums[t] += partialSums[t+ 2];
        if (id <  1)
            partialSums[t] += partialSums[t+ 1];

        // Write result
        if (id == 0)
        {
            out[myRow] = partialSums[t];
        }
    }
```

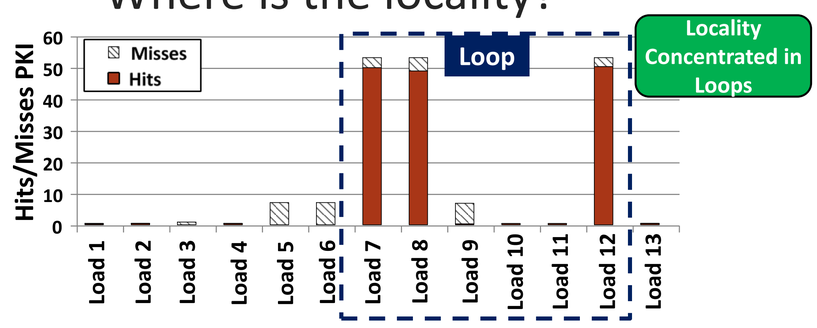
Explicit Scratchpad Use

Dependent on Warp Size

Parallel Reduction

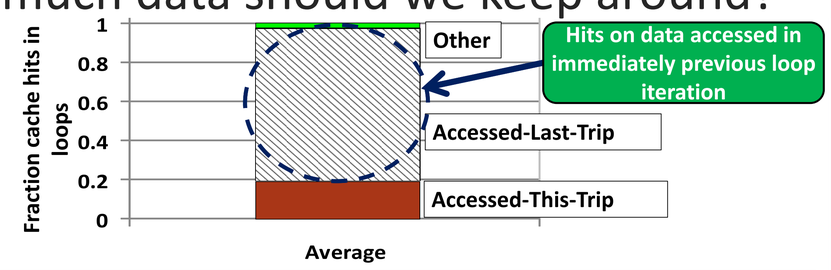
Cache Footprint Prediction

Where is the locality?



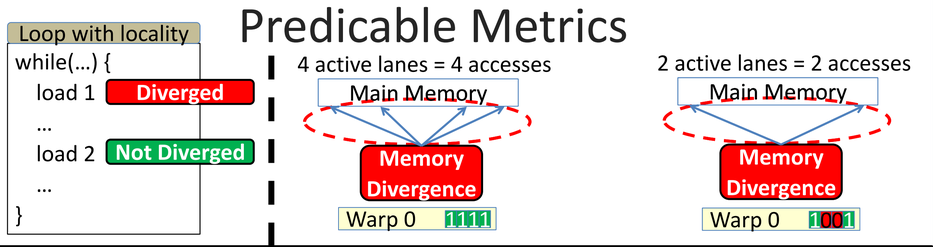
Create a cache footprint for each warp in loop

How much data should we keep around?



Cache footprint = Number of lines accessed in one loop iteration

Predicable Metrics

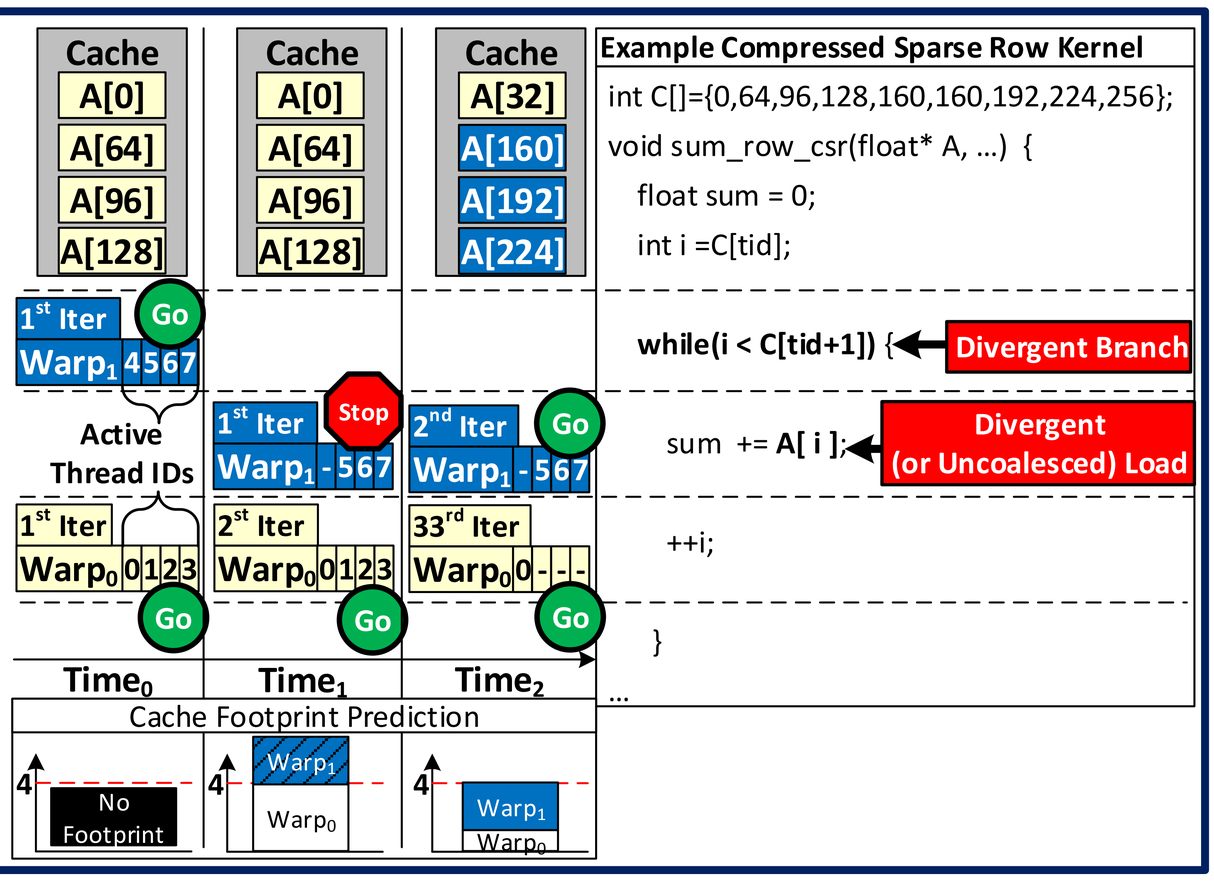


Static loads: Either diverged or not diverged

Active Mask: Determines lines accessed by divergent loads

Example Operation

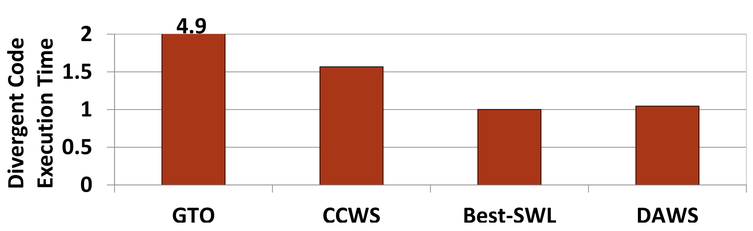
Cache is 4 entries, 128B lines and fully associative.
By $Time_0$, warp 0 has entered loop and loaded 4 lines into cache. By $Time_1$, warp 0 has captured spatial locality, DAWS measures footprint. Warp 1 is prevented from scheduling as DAWS predicts it will oversubscribe cache. By $Time_2$, warp 0 has accessed 4 lines for 32 iterations and loaded 1 new line. 3 lanes have exited loop, decreasing footprint. Warp 1 and warp 0 are allowed to capture spatial locality together.



Results

Programmability Case Study Results

Diverged Code vs. Locality Managed Code



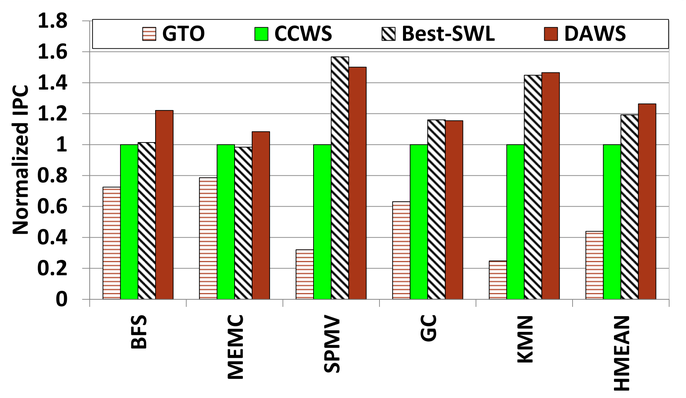
With DAWS: Divergent Code within 4% of Locality Managed Code with no Programmer Input

- Greedy-then-oldest (GTO)
- Cache-Conscious Wavefront Scheduling (CCWS)
- Profile based Static Wavefront Limiting (Best-SWL)
- **Divergence-Aware Warp Scheduling (DAWS)**

Proactive Predictions: Reduce Cache Misses

Branch Divergence Awareness: Increases Multithreading when Appropriate

Performance vs. Other Schedulers on Cache-Sensitive Applications



26% Speedup over Cache-Conscious Wavefront Scheduling

Timothy G. Rogers[1], Mike O'Connor[2], Tor M. Aamodt[1]
[1]The University of British Columbia [2]NVIDIA Research

MICRO 2013. Davis, CA