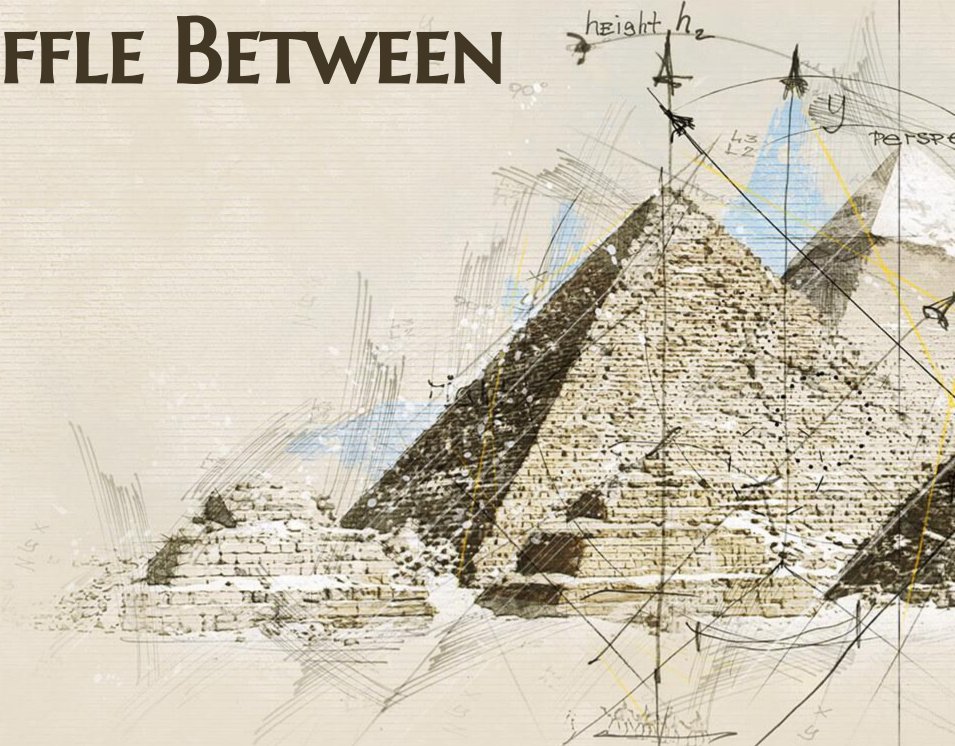
Supporting Data Shuffle Between Threads in OpenMP

Anjia Wang, Xinyao Yi, Yonghong Yan

University of North Carolina at Charlotte - USA

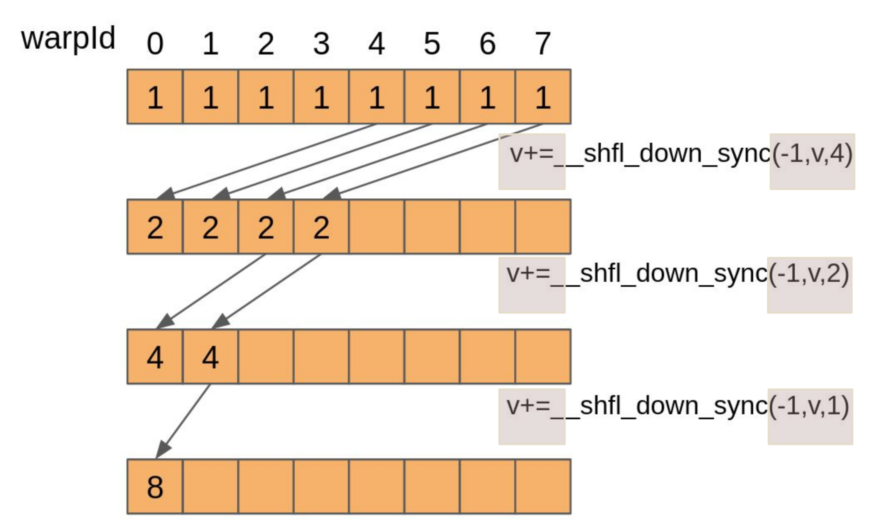


ANGEDA

- Motivation
- Using shuffle in OpenMP Runtime
    - **reduction clause**
- Proposed shuffle directive and clause
    - **2D Stencil**
- Experimental results
- Related work
- Conclusion and future work

Motivation

- NVIDIA GPU shuffle instruction
  - __shfl_up_sync, __shfl_down_sync, …
- AMD GPU cross-lane operations
  - ds_permute_32, ds_bpermute_b32
- Shuffle between SIMD/vector lanes
  - Intel: SHUFPS, VSHUFPS, …



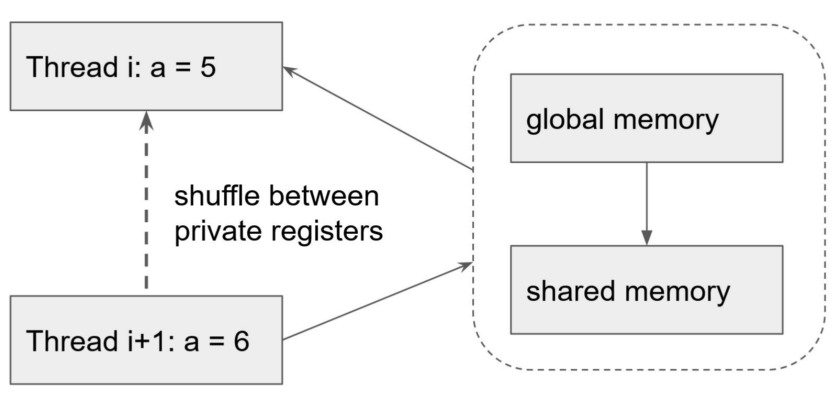
v+=__shfl_down_sync(-1,v,4)

v+=__shfl_down_sync(-1,v,2)

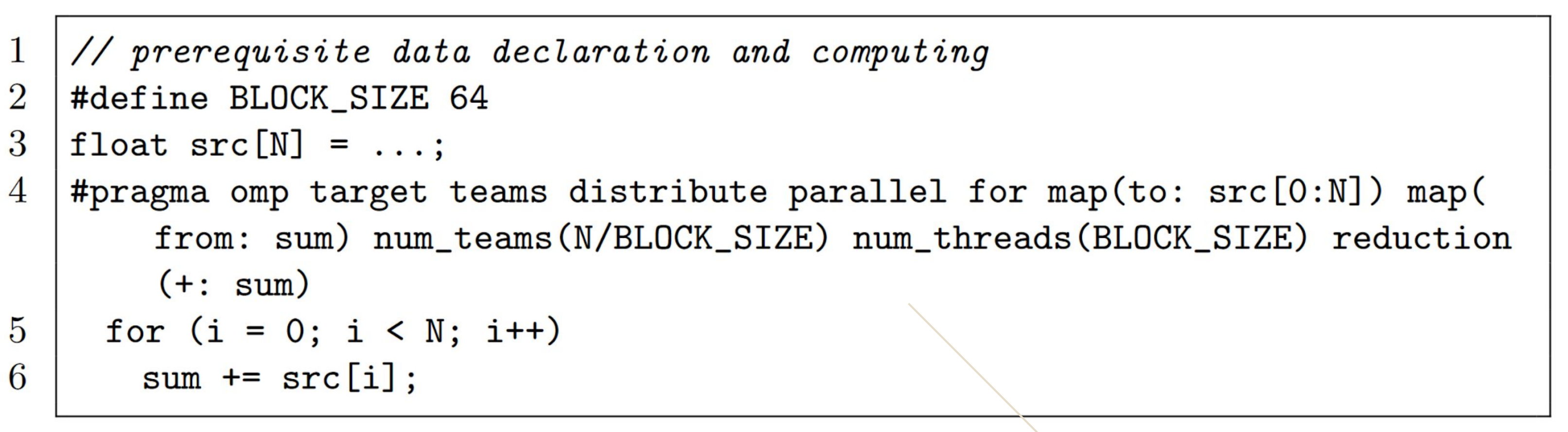
v+=__shfl_down_sync(-1,v,1)

Motivation

- Sharing data between two threads
  - Read **a** from T(i+1) to T(i)
- Not using shuffle
  - Transfer via global memory/shared memory
- Using shuffle
  - Directly copy from the register of T(i+1)



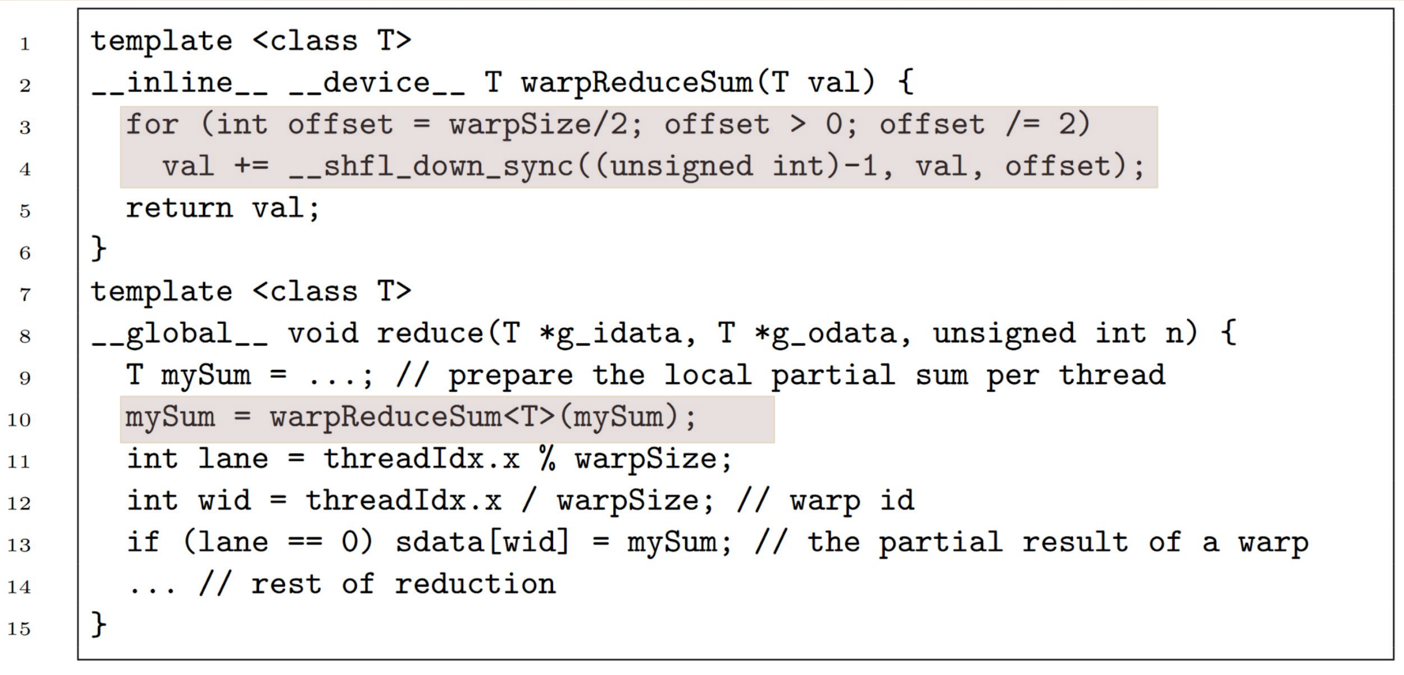
Runtime implementation of Reduction clause



```
// prerequisite data declaration and computing
#define BLOCK_SIZE 64
float src[N] = ...;
#pragma omp target teams distribute parallel for map(to: src[0:N]) map(
    from: sum) num_teams(N/BLOCK_SIZE) num_threads(BLOCK_SIZE) reduction
    (+: sum)
  for (i = 0; i < N; i++)
    sum += src[i];
```

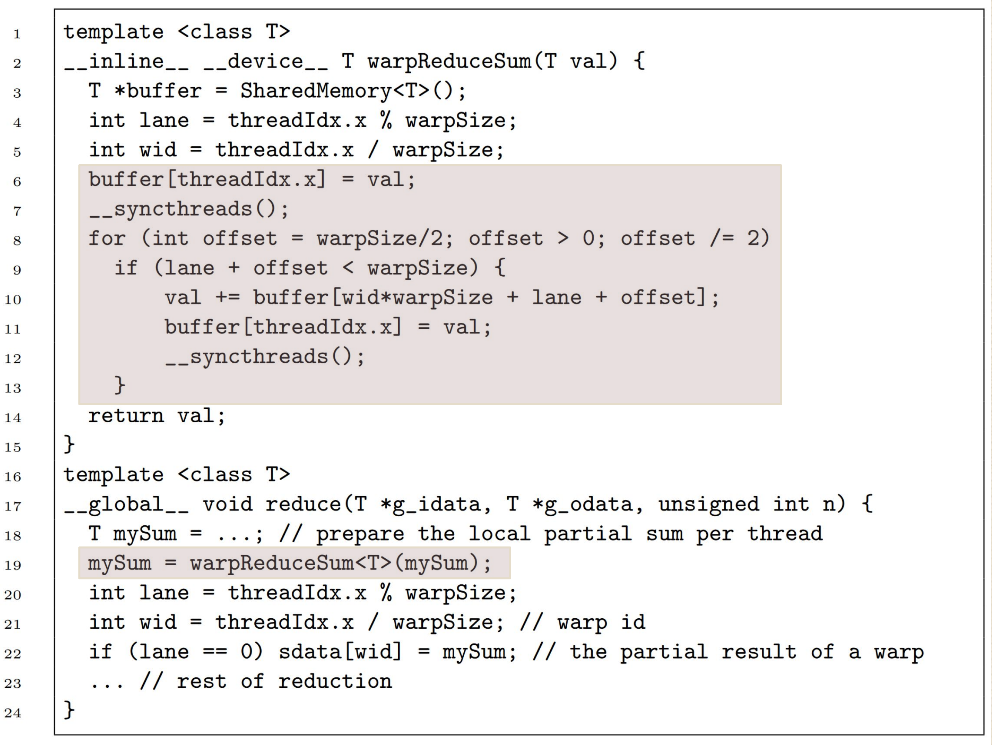
- Four versions are implemented:
  - Using global memory, shared memory, shared memory simulated shuffle, and native shuffle.
- Clang/LLVM 10.1 is used as reference.

Runtime implementation of Reduction clause



```
template <class T>
__inline__ __device__ T warpReduceSum(T val) {
  for (int offset = warpSize/2; offset > 0; offset /= 2)
    val += __shfl_down_sync((unsigned int)-1, val, offset);
  return val;
}
template <class T>
__global__ void reduce(T *g_idata, T *g_odata, unsigned int n) {
  T mySum = ...; // prepare the local partial sum per thread
  mySum = warpReduceSum<T>(mySum);
  int lane = threadIdx.x % warpSize;
  int wid = threadIdx.x / warpSize; // warp id
  if (lane == 0) sdata[wid] = mySum; // the partial result of a warp
  ... // rest of reduction
}
```

Runtime implementation of Reduction clause



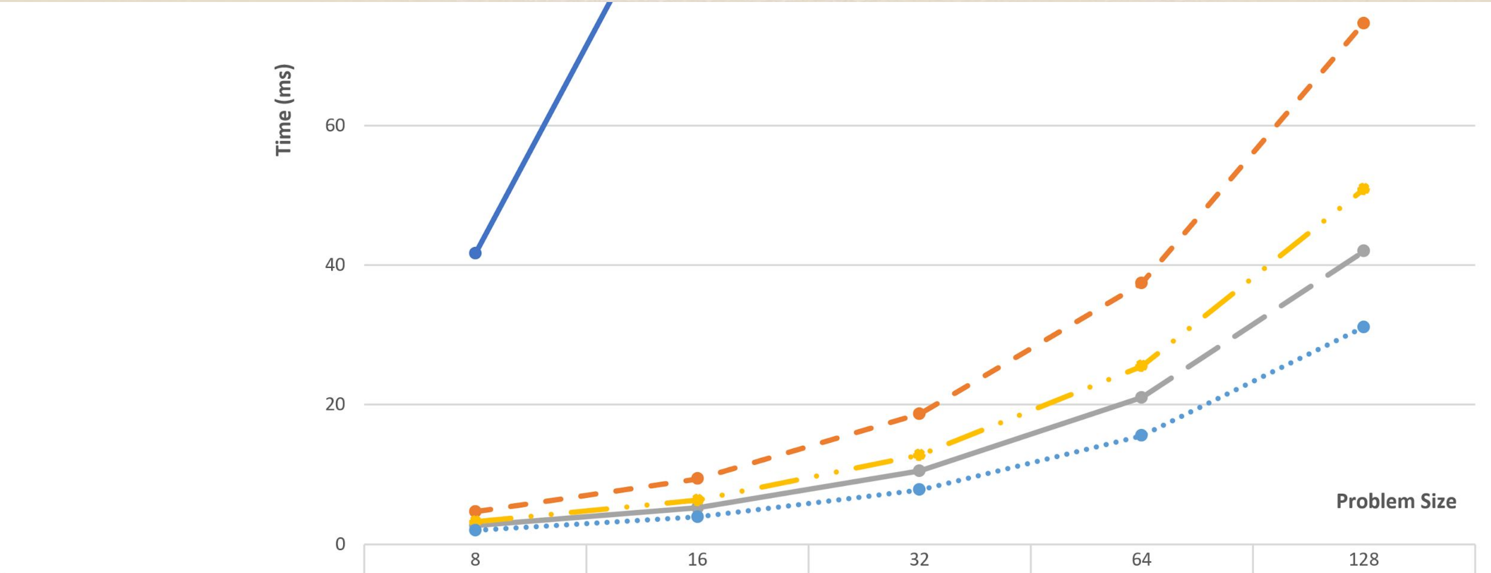
```
template <class T>
__inline__ __device__ T warpReduceSum(T val) {
  T *buffer = SharedMemory<T>();
  int lane = threadIdx.x % warpSize;
  int wid = threadIdx.x / warpSize;
  buffer[threadIdx.x] = val;
  __syncthreads();
  for (int offset = warpSize/2; offset > 0; offset /= 2)
    if (lane + offset < warpSize) {
          val += buffer[wid*warpSize + lane + offset];
          buffer[threadIdx.x] = val;
          __syncthreads();
    }
  return val;
}
template <class T>
__global__ void reduce(T *g_idata, T *g_odata, unsigned int n) {
  T mySum = ...; // prepare the local partial sum per thread
  mySum = warpReduceSum<T>(mySum);
  int lane = threadIdx.x % warpSize;
  int wid = threadIdx.x / warpSize; // warp id
  if (lane == 0) sdata[wid] = mySum; // the partial result of a warp
  ... // rest of reduction
}
```

- On the platform that doesn’t support shuffle instruction, we can simulate it using shared memory for better portability.
- At runtime, different library could be linked to the same interface.

Experimental environment

- Hardware:
  - Intel Xeon E5-2699 V3 (18 cores) * 2, 256 GB RAM, NVIDIA Tesla K80 24GB
  - Intel Xeon W-2133 (12 cores), 32 GB RAM, NVIDIA Quadro P400 2GB
- Software:
  - Ubuntu 18.04 LTS
  - CUDA SDK 10.2
  - Clang/LLVM 10.1 for OpenMP offloading

Runtime implementation of Reduction clause



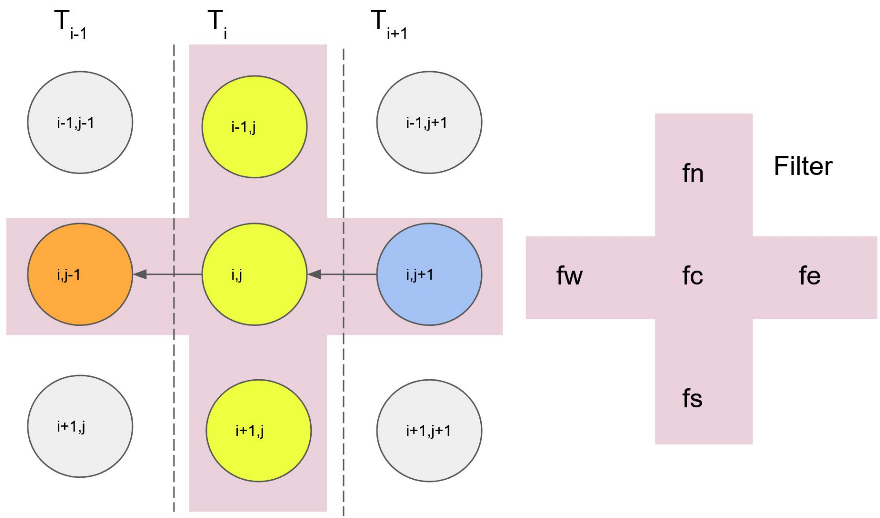
	8	16	32	64	128
OpenMP	41.672	101.3	154.62	266.95	528.96
Global Memory	4.67	9.36	18.69	37.38	74.69
Shared Memory	2.68	5.27	10.52	21.02	41.99
Shared-Memory Simulated Shuffle	3.23	6.37	12.76	25.48	50.86
Native Shuffle	1.98	3.93	7.79	15.56	31.13

Proposed shuffle extension in OpenMP

- **shuffle clause**: used with parallel or teams directive to declare shuffling variables.
  Syntax: shuffle (*variable-list*)
- **shuffle directive**: an executive directive to specify when and how the data should be shuffled.
  Syntax: #pragma omp shuffle *clause*
  clause: sync/up/down (*mask-modifier*[,] *src-modifier*[,] *dst-variable* [*operator*], *shuffle-variable*)
- shuffle up (-1, 1, a, a) // By default, the operator is “=”.
  shuffle down (-1, 2, b +, b)

2D 5-point Stencil

- Stencil operation applies a filter to each point.
- Given a cross-shape filter, to compute the point (i, j), three threads T(i-1), T(i), and T(i+1) are involved.
- Each thread computes one column of the filter and passes the partial result to its neighbour except the T(i-1).



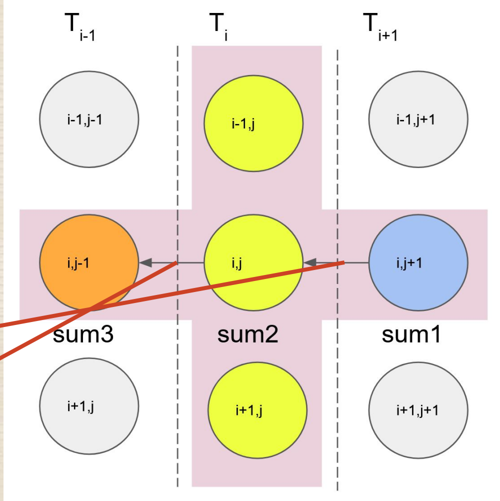
result(i,j) = p(i,j+1)*fe

+ p(i-1,j)*fn + p(i,j)*fc + p(i+1,j)*fs

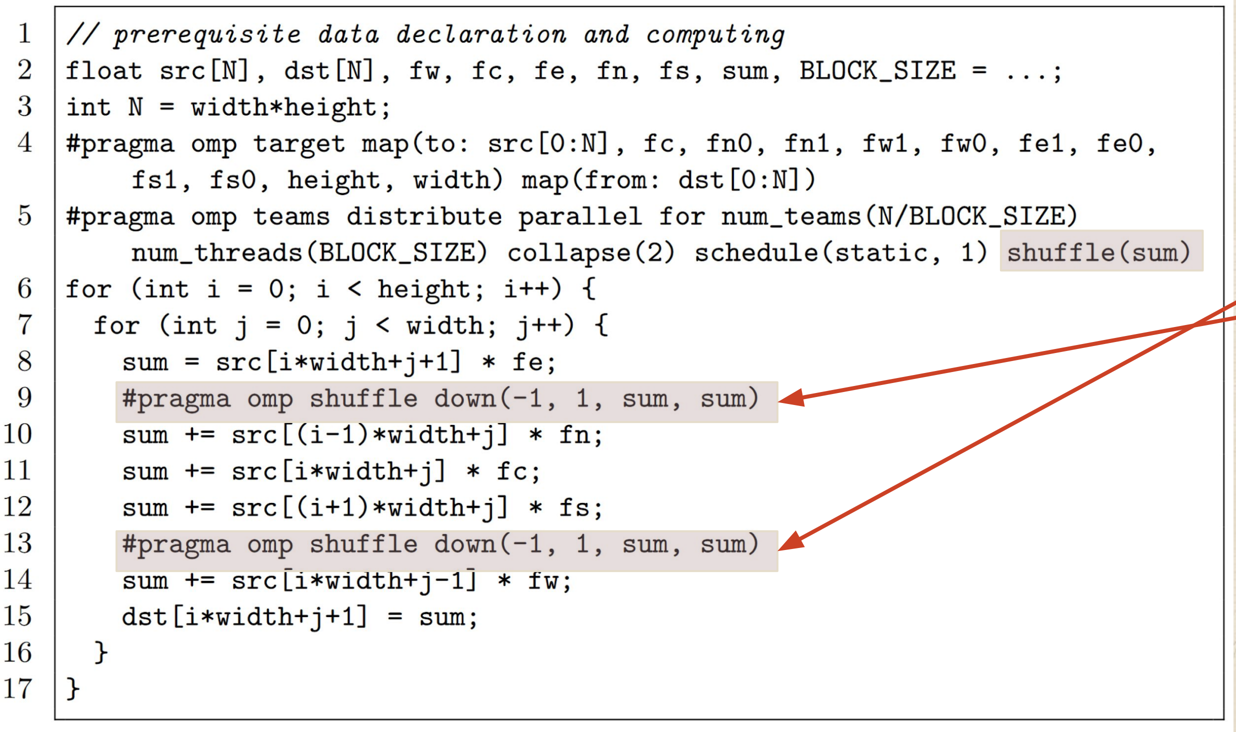
+ p(i,j-1)*fw

Using shuffle constructs in 2D stencil

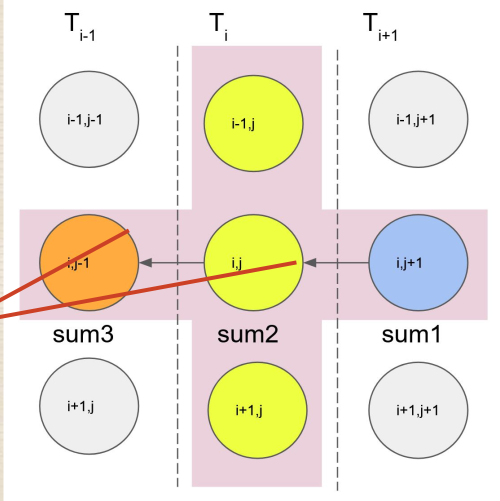
```
// prerequisite data declaration and computing
float src[N], dst[N], fw, fc, fe, fn, fs, sum, BLOCK_SIZE = ...;
#pragma omp target teams map(to: src[0:N], fw, fc, fe, fn, fs) map(from:
    dst[0:N]) num_teams(N/BLOCK_SIZE)
#pragma omp parallel num_threads(BLOCK_SIZE) shuffle(sum) // declare sum
    for shuffle
    {   // prepared needed data, such as global index of src item and dst
         item
        int global_index[3], index = ...;
        sum = src[global_index[1]] * fe; // partial sum1
        #pragma omp shuffle down(-1, 1, sum, sum) // thread n shuffles sum
             from thread n+1 and replace its own sum copy
        sum += src[global_index[0]] * fn;
        sum += src[global_index[1]] * fc;
        sum += src[global_index[2]] * fs; // partial sum2
        #pragma omp shuffle down(-1, 1, sum, sum)
        sum += src[global_index[1]] * fw; // partial sum3
        dst[index] = sum; // write the final result to output array dst
    }
```



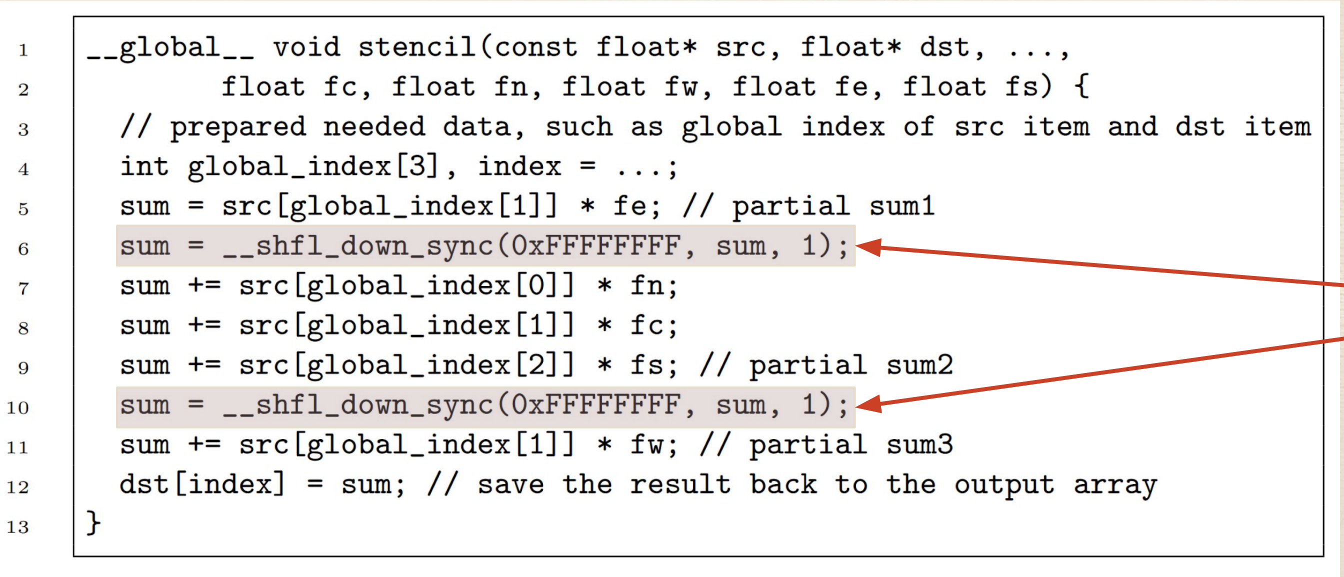
Using shuffle constructs in 2D stencil with worksharing



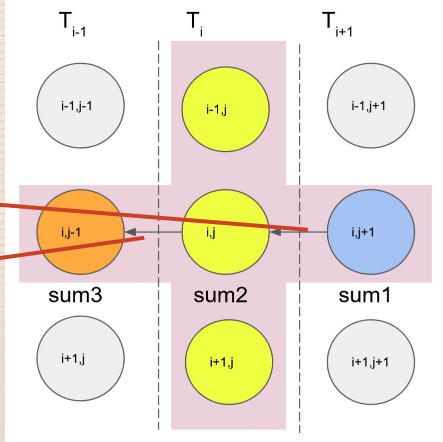
```
// prerequisite data declaration and computing
float src[N], dst[N], fw, fc, fe, fn, fs, sum, BLOCK_SIZE = ...;
int N = width*height;
#pragma omp target map(to: src[0:N], fc, fn0, fn1, fw1, fw0, fe1, fe0,
     fs1, fs0, height, width) map(from: dst[0:N])
#pragma omp teams distribute parallel for num_teams(N/BLOCK_SIZE)
     num_threads(BLOCK_SIZE) collapse(2) schedule(static, 1) shuffle(sum)
for (int i = 0; i < height; i++) {
  for (int j = 0; j < width; j++) {
    sum = src[i*width+j+1] * fe;
    #pragma omp shuffle down(-1, 1, sum, sum)
    sum += src[(i-1)*width+j] * fn;
    sum += src[i*width+j] * fc;
    sum += src[(i+1)*width+j] * fs;
    #pragma omp shuffle down(-1, 1, sum, sum)
    sum += src[i*width+j-1] * fw;
    dst[i*width+j+1] = sum;
  }
}
```



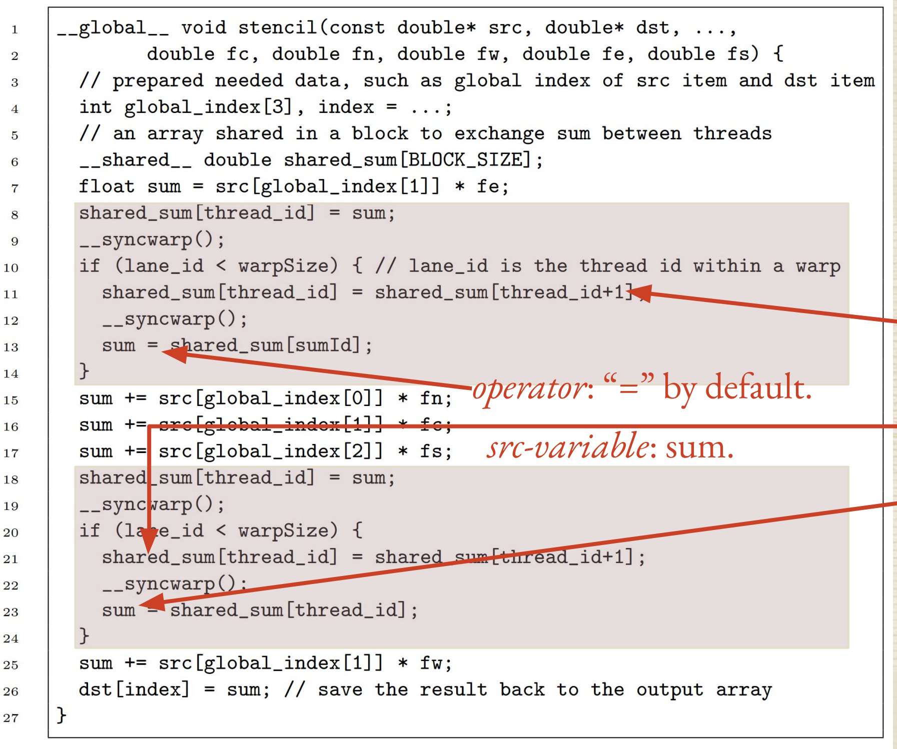
IMPLEMENTATION USING NATIVE SHUFFLE INSTRUCTION



```
__global__ void stencil(const float* src, float* dst, ...,
        float fc, float fn, float fw, float fe, float fs) {
  // prepared needed data, such as global index of src item and dst item
  int global_index[3], index = ...;
  sum = src[global_index[1]] * fe; // partial sum1
  sum = __shfl_down_sync(0xFFFFFFFF, sum, 1);
  sum += src[global_index[0]] * fn;
  sum += src[global_index[1]] * fc;
  sum += src[global_index[2]] * fs; // partial sum2
  sum = __shfl_down_sync(0xFFFFFFFF, sum, 1);
  sum += src[global_index[1]] * fw; // partial sum3
  dst[index] = sum; // save the result back to the output array
}
```



IMPLEMENTATION USING SIMULATED SHUFFLE INSTRUCTION



```
__global__ void stencil(const double* src, double* dst, ...,
        double fc, double fn, double fw, double fe, double fs) {
  // prepared needed data, such as global index of src item and dst item
  int global_index[3], index = ...;
  // an array shared in a block to exchange sum between threads
  __shared__ double shared_sum[BLOCK_SIZE];
  float sum = src[global_index[1]] * fe;
  shared_sum[thread_id] = sum;
  __syncwarp();
  if (lane_id < warpSize) { // lane_id is the thread id within a warp
    shared_sum[thread_id] = shared_sum[thread_id+1];
    __syncwarp();
    sum = shared_sum[sumId];
  }
  sum += src[global_index[0]] * fn;
  sum += src[global_index[1]] * fc;
  sum += src[global_index[2]] * fs;
  shared_sum[thread_id] = sum;
  __syncwarp();
  if (lane_id < warpSize) {
    shared_sum[thread_id] = shared_sum[thread_id+1];
    __syncwarp();
    sum = shared_sum[thread_id];
  }
  sum += src[global_index[1]] * fw;
  dst[index] = sum; // save the result back to the output array
}
```

- Buffer in shared memory
- Each team member has a spot

src-modifier: thread id offset is 1.

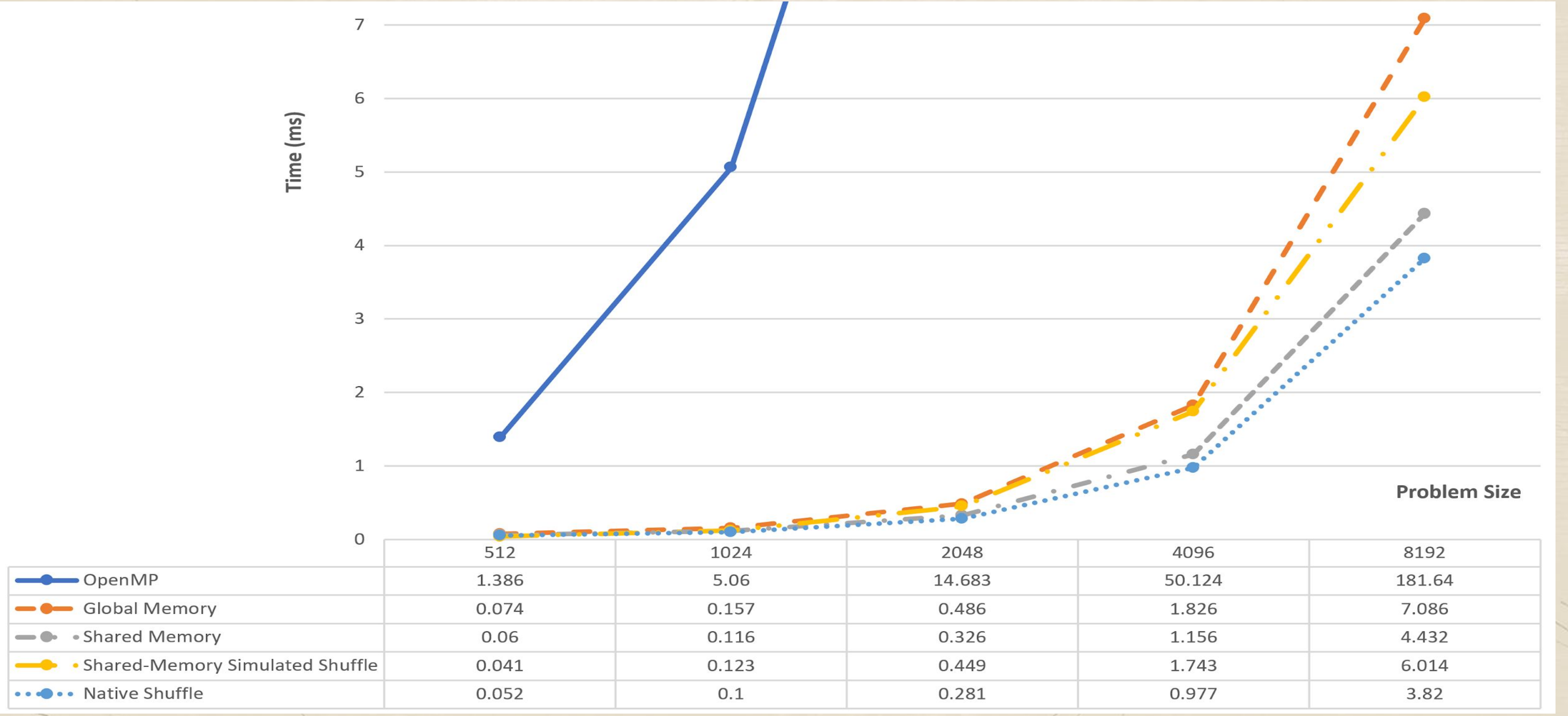
operator: “=” by default.

- shuffle down (-1, 1, sum, sum)

src-variable: sum.

dst-variable: sum.

Performance comparison of 2D stencil



	512	1024	2048	4096	8192
OpenMP	1.386	5.06	14.683	50.124	181.64
Global Memory	0.074	0.157	0.486	1.826	7.086
Shared Memory	0.06	0.116	0.326	1.156	4.432
Shared-Memory Simulated Shuffle	0.041	0.123	0.449	1.743	6.014
Native Shuffle	0.052	0.1	0.281	0.977	3.82

Related Work

- Liu and Schmit (2015) use warp shuffle functions in a similar way to develop LightSpMV, which is a faster algorithm of sparse matrix-vector multiplication.
- Tangram is a high-level programming framework for GPU programming and it uses atomic and shuffle functions (Gonzalo et al., 2019).
  - Compiler inserts shuffle instruction for loop optimization
- With the help of shuffle instructions, Chen et al. (2019) realize the systolic execution on GPU and demonstrate superior performance for 2D stencil in CUDA than most of state-of-the-art implementations.

Conclusion

- **Runtime usage of shuffle and OpenMP extension for shuffle**
  - Users can use shuffle in a high-level programming model
  - Our implementation can obtain up to 25x speed up over LLVM standard OpenMP library, and 2.39x speed up over other hand-written highly optimized versions.
- **Ongoing/future work**
  - Exploration to using shuffle in SIMD directive

Thanks!

Any questions?

This work was supported by the National Science Foundation under Grant No. 2015254 and 1409946.