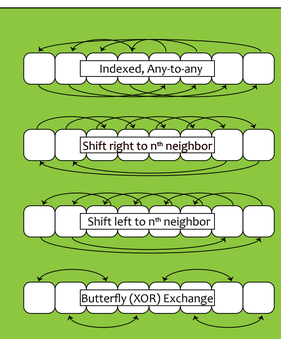
Enabling Efficient Intra-Warp Communication for Fourier Transforms in a Many-Core Architecture

Student:
Carlo del Mundo (cdel@vt.edu)

Faculty:
Wu-chun Feng (feng@cs.vt.edu)

This work was supported in part by NSF I/UCRC IIP-0804155 via the NSF Center for High-Performance Reconfigurable Computing

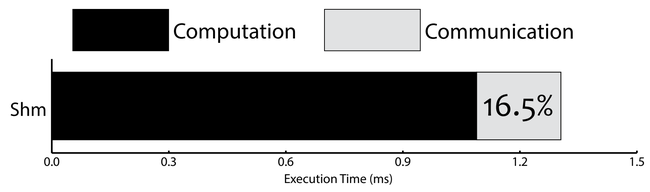


Shuffle Access Patterns

Goal >> *Accelerate intra-warp communication for FFT using shuffle instructions* **<<**

1. Motivation

- Communication oftentimes contributes to the cost of an algorithm.



Execution profile of a 256-pt FFT

- Fraction$_{enhanced}$ = 16.5%
- Max. Speedup[1] = 1.19x
- Achieved Speedup[2] = 1.17x

[1] At 37.5% occupancy; [2] At 50% occupancy

- A new hardware mechanism (shuffle) allows for fast, intra-warp communication without shared memory. However, there is little success in accelerating applications using shuffle outside of vendor claims.

2. Background

- **Shared memory**: traditional memory space for scratchpad communication for threads in a threadblock.
- **Shuffle**: a new communication mechanism that allows for intra-warp register to register exchange. It was introduced in NVIDIA SM 3.0+. It supports arbitrary indexed references for data reads within a warp and has the advantage of carrying out a store-and-load operation in one step [1].
- CUDA **local memory**: a per-thread spill-over memory resident in the L1 cache or global memory. Accessing CUDA local memory is slow.
- CUDA **SELP instructions:** PTX instructions that allow for predicated stores. Generally, these instructions reduce (but do not eliminate) divergence. Instruction are of the form: a = (predicate) ? b : c;

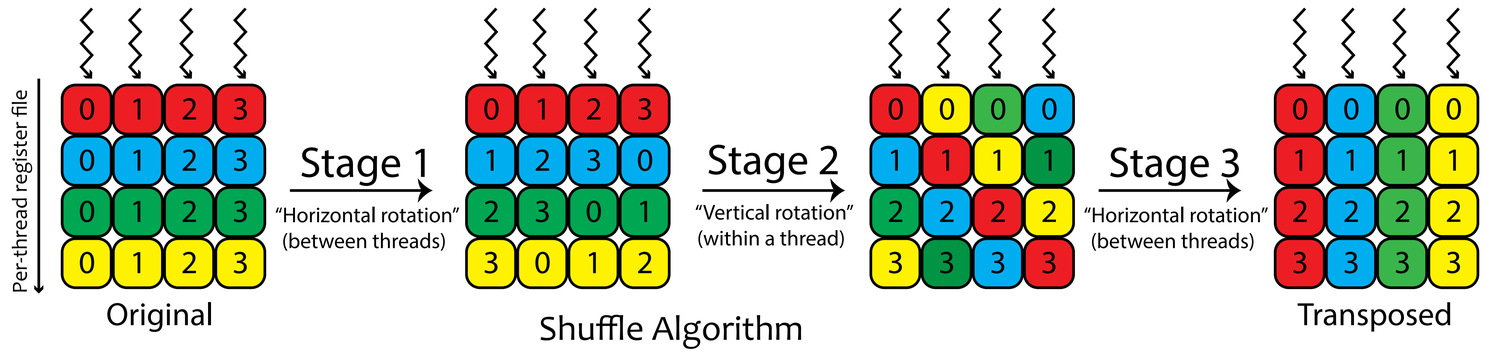
[1] NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ Compute Architecture: Kepler™ v1.0, Available <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>.

3. Approach

- Apply the shuffle mechanism to accelerate matrix transpose. Matrix transpose is the fundamental communication stage in the Fast Fourier Transform (FFT).

Application & Experimental Testbed

- Batched 256-pt FFT (128 MB of data). Each FFT contains a transpose 16x16 in size.
- NVIDIA Tesla K20c, 8192 thread blocks, 64 threads per block
- Our algorithm is shown for a 4x4 transpose. It is composed of horizontal and vertical rotations. Note: our algorithm is scalable up to 32x32 in size.



Shuffle Algorithm

4. Characterization

- Kernels shown: 4x4 transpose. Actual results use the 16x16 transpose versions embedded inside a 256-pt FFT kernel
- We divide our implementations between the shared memory version ("shm") and several incremental shuffle versions ("naive", "div", "selp ip", "selp oop").

4.1 Shared Memory

Shm

- Traditional mechanism to perform a transpose operation for threads in a threadblock.

```
#define WIDTH 4
void __global__ shm(int *input_ary, int *output_ary)
{
 int src_registers[4];
 extern __shared__ int shared[];
 int *shared_column_stride = shared + batch_id * 16 + threadIdx.x;
 int *shared_row_stride = shared + batch_id * 16 + 4 * threadIdx.x;

 // Initialization + Load from global memory to registers

  for (int i = 0; i < 4; ++i)
    shared_column_stride[4*i] = registers[i];
  for (int i = 0; i < 4; ++i)
    registers[i] = shared_row_stride[i];

 // Store from registers to global memory
}
```

4.2 Shuffle

- Shuffle implementations bottlenecked by stage two (vertical rotations). This step is responsible for rearranging data within a thread's private register data.
- Our optimization process focuses on accelerating stage two.

Naive

- CUDA local memory allocation and usage causes poor performance

```
#define WIDTH 4
void __global__ shuf_naive(int *input_ary, int *output_ary)
{
 int src_registers[4];

 // Initialization + Load from global memory to registers

 // <!-- START: Stage One (Horizontal Rotations) -->
 int addr = threadIdx.x % 4;
 addr = __shfl(addr, (tid + 1) % 4, WIDTH);
 src_registers[1] = __shfl(src_registers[1], addr, WIDTH);
 addr = __shfl(addr, (tid + 1) % 4, WIDTH);
 src_registers[2] = __shfl(src_registers[2], addr, WIDTH);
 addr = __shfl(addr, (tid + 1) % 4, WIDTH);
 src_registers[3] = __shfl(src_registers[3], addr, WIDTH);
 // <!-- END: Stage One (Horizontal Rotations) -->

 // <!-- START: Stage Two (Vertical Rotations) -->
 for (int k = 0; k < 4; ++k)
      dst_registers[k] = src_registers[(4 - tid + k) % 4];
 // <!-- END: Stage Two (Vertical Rotations) -->

 // <!-- START: Stage Three (Horizontal Rotations) -->
 addr = (4 - tid) % 4;
 src_registers[0] = __shfl(src_registers[0], addr, WIDTH);
 addr = __shfl(addr, (tid + 3) % 4, WIDTH);
 src_registers[1] = __shfl(src_registers[1], addr, WIDTH);
 addr = __shfl(addr, (tid + 3) % 4, WIDTH);
 src_registers[2] = __shfl(src_registers[2], addr, WIDTH);
 addr = __shfl(addr, (tid + 3) % 4, WIDTH);
 src_registers[3] = __shfl(src_registers[3], addr, WIDTH);
 // <!-- END: Stage Three (Horizontal Rotations) -->

 // Store from registers to global memory
}
```

Offending code. Compiler is forced to place registers into CUDA local memory since indices cannot be determined at compile time

Div

- Refactored stage two by unrolling (but introduces divergence)
- No CUDA local memory

```
#define WIDTH 4
void __global__ shuf_div(int *input_ary, int *output_ary)
{
 int src_registers[4];

 // Initialization + Load from global memory to registers

 // <!-- START: Stage One (Horizontal Rotations) -->
 int addr = threadIdx.x % 4;
 addr = __shfl(addr, (tid + 1) % 4, WIDTH);
 src_registers[1] = __shfl(src_registers[1], addr, WIDTH);
 addr = __shfl(addr, (tid + 1) % 4, WIDTH);
 src_registers[2] = __shfl(src_registers[2], addr, WIDTH);
 addr = __shfl(addr, (tid + 1) % 4, WIDTH);
 src_registers[3] = __shfl(src_registers[3], addr, WIDTH);
 // <!-- END: Stage One (Horizontal Rotations) -->

 // <!-- START: Stage Two (Vertical Rotations) -->
 int tmp = src_registers[0];
 if (tid == 1)
 {
      src_registers[0] = src_registers[3];
      src_registers[3] = src_registers[2];
      src_registers[2] = src_registers[1];
      src_registers[1] = tmp;
 }
 else if (tid == 2)
 {
      src_registers[0] = src_registers[2];
      src_registers[2] = tmp;
      tmp = src_registers[1];
      src_registers[1] = src_registers[3];
      src_registers[3] = tmp;
 }
 else if (tid == 3)
 {
      src_registers[0] = src_registers[1];
      src_registers[1] = src_registers[2];
      src_registers[2] = src_registers[3];
      src_registers[3] = tmp;
 }
 // <!-- END: Stage Two (Vertical Rotations) -->

 // <!-- START: Stage Three (Horizontal Rotations) -->
 addr = (4 - tid) % 4;
 src_registers[0] = __shfl(src_registers[0], addr, WIDTH);
 addr = __shfl(addr, (tid + 3) % 4, WIDTH);
 src_registers[1] = __shfl(src_registers[1], addr, WIDTH);
 addr = __shfl(addr, (tid + 3) % 4, WIDTH);
 src_registers[2] = __shfl(src_registers[2], addr, WIDTH);
 addr = __shfl(addr, (tid + 3) % 4, WIDTH);
 src_registers[3] = __shfl(src_registers[3], addr, WIDTH);
 // <!-- END: Stage Three (Horizontal Rotations) -->

 // Store from registers to global memory
}
```

No CUDA local memory allocation, but this introduces divergence.

SELP (IP)

- Replaced divergence with SELP
- No CUDA local memory

```
#define WIDTH 4
void __global__ shuf_selp_ip(int *input_ary, int *output_ary)
{
 int src_registers[4];

 // Initialization + Load from global memory to registers

 // <!-- START: Stage One (Horizontal Rotations) -->
 int tid = threadIdx.x % 4;
 int addr = tid;
 addr = __shfl(addr, (tid + 1) % 4, WIDTH);
 src_registers[1] = __shfl(src_registers[1], addr, WIDTH);
 addr = __shfl(addr, (tid + 1) % 4, WIDTH);
 src_registers[2] = __shfl(src_registers[2], addr, WIDTH);
 addr = __shfl(addr, (tid + 1) % 4, WIDTH);
 src_registers[3] = __shfl(src_registers[3], addr, WIDTH);
 // <!-- END: Stage One (Horizontal Rotations) -->

 // <!-- START: Stage Two (Vertical Rotations) -->
 int tmp = src_registers[0];
 src_registers[0] = (tid == 1) ? src_registers[3] : src_registers[0];
 src_registers[3] = (tid == 1) ? src_registers[2] : src_registers[3];
 src_registers[2] = (tid == 1) ? src_registers[1] : src_registers[2];
 src_registers[1] = (tid == 1) ? tmp : src_registers[1];

 tmp = src_registers[0];
 src_registers[0] = (tid == 2) ? src_registers[2] : src_registers[0];
 src_registers[2] = (tid == 2) ? tmp : src_registers[2];

 tmp = src_registers[1];
 src_registers[1] = (tid == 2) ? src_registers[3] : src_registers[1];
 src_registers[3] = (tid == 2) ? tmp : src_registers[3];

 tmp = src_registers[0];
 src_registers[0] = (tid == 3) ? src_registers[1] : src_registers[0];
 src_registers[1] = (tid == 3) ? src_registers[2] : src_registers[1];
 src_registers[2] = (tid == 3) ? src_registers[3] : src_registers[2];
 src_registers[3] = (tid == 3) ? tmp : src_registers[3];
 // <!-- END: Stage Two (Vertical Rotations) -->

 // <!-- START: Stage Three (Horizontal Rotations) -->
 addr = (4 - tid) % 4;
 src_registers[0] = __shfl(src_registers[0], addr, WIDTH);
 addr = __shfl(addr, (tid + 3) % 4, WIDTH);
 src_registers[1] = __shfl(src_registers[1], addr, WIDTH);
 addr = __shfl(addr, (tid + 3) % 4, WIDTH);
 src_registers[2] = __shfl(src_registers[2], addr, WIDTH);
 addr = __shfl(addr, (tid + 3) % 4, WIDTH);
 src_registers[3] = __shfl(src_registers[3], addr, WIDTH);
 // <!-- END: Stage Three (Vertical Rotations) -->

 // Store from registers to global memory
}
```

Predicated stores (SELP instructions) are more efficient than if, else if clauses.

SELP (OOP)

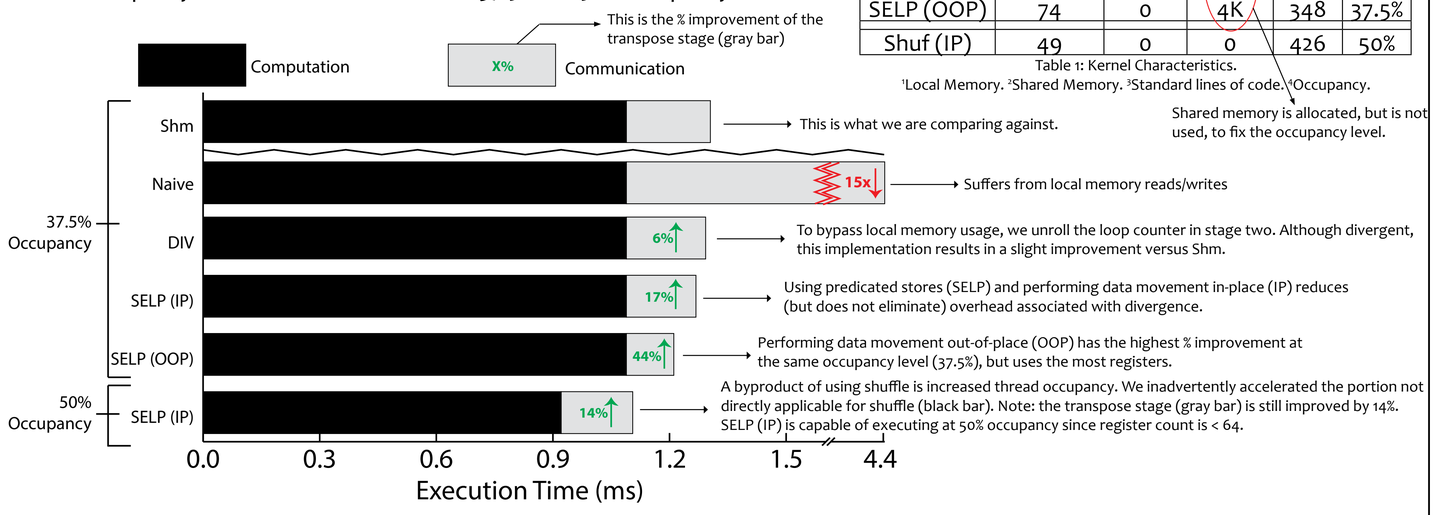
- Perform movement out of place
- No CUDA local memory

```
#define WIDTH 4
void __global__ shuf_selp_oop(int *input_ary, int *output_ary)
{
 int src_registers[4];
 int dst_registers[4];

 // Initialization + Load from global memory to registers

 // <!-- START: Stage One (Horizontal Rotations) -->
 int tid = threadIdx.x % 4;
 int addr = tid;
 addr = __shfl(addr, (tid + 1) % 4, WIDTH);
 src_registers[1] = __shfl(src_registers[1], addr, WIDTH);
 addr = __shfl(addr, (tid + 1) % 4, WIDTH);
 src_registers[2] = __shfl(src_registers[2], addr, WIDTH);
 addr = __shfl(addr, (tid + 1) % 4, WIDTH);
 src_registers[3] = __shfl(src_registers[3], addr, WIDTH);
 // <!-- END: Stage One (Horizontal Rotations) -->

 // <!-- START: Stage Two (Vertical Rotations) -->
 dst_registers[0] = (tid == 0) ? src_registers[0] : dst_registers[0];
 dst_registers[1] = (tid == 0) ? src_registers[1] : dst_registers[1];
 dst_registers[2] = (tid == 0) ? src_registers[2] : dst_registers[2];
 dst_registers[3] = (tid == 0) ? src_registers[3] : dst_registers[3];

 dst_registers[0] = (tid == 1) ? src_registers[3] : dst_registers[0];
 dst_registers[1] = (tid == 1) ? src_registers[0] : dst_registers[1];
 dst_registers[2] = (tid == 1) ? src_registers[1] : dst_registers[2];
 dst_registers[3] = (tid == 1) ? src_registers[2] : dst_registers[3];

 dst_registers[0] = (tid == 2) ? src_registers[2] : dst_registers[0];
 dst_registers[1] = (tid == 2) ? src_registers[3] : dst_registers[1];
 dst_registers[2] = (tid == 2) ? src_registers[0] : dst_registers[2];
 dst_registers[3] = (tid == 2) ? src_registers[1] : dst_registers[3];

 dst_registers[0] = (tid == 3) ? src_registers[1] : dst_registers[0];
 dst_registers[1] = (tid == 3) ? src_registers[2] : dst_registers[1];
 dst_registers[2] = (tid == 3) ? src_registers[3] : dst_registers[2];
 dst_registers[3] = (tid == 3) ? src_registers[0] : dst_registers[3];
 // <!-- END: Stage Two (Vertical Rotations) -->

 // <!-- START: Stage Three (Horizontal Rotations) -->
 addr = (4 - tid) % 4;
 src_registers[0] = __shfl(dst_registers[0], addr, WIDTH);
 addr = __shfl(addr, (tid + 3) % 4, WIDTH);
 src_registers[1] = __shfl(dst_registers[1], addr, WIDTH);
 addr = __shfl(addr, (tid + 3) % 4, WIDTH);
 src_registers[2] = __shfl(dst_registers[2], addr, WIDTH);
 addr = __shfl(addr, (tid + 3) % 4, WIDTH);
 src_registers[3] = __shfl(dst_registers[3], addr, WIDTH);
 // <!-- END: Stage Three (Vertical Rotations) -->

 // Store from registers to global memory
}
```

Performing out of place (OOP) is faster, but uses more registers.

5. Results

- Traditional shared memory FFTs are limited to 37.5% occupancy
- Shuffle implementations allocate, but do not use, shared memory to maintain a fixed occupancy. Results shown are for both 37.5% and 50% occupancy levels.



Name	Registers	LMEM[1] (bytes)	Shm[2] (bytes)	SLOC[3]	OCC[4]
Shared	50	0	4K	11	37.5%
Naive	52	128	4K	100	37.5%
Div	72	0	4K	462	37.5%
SELP (IP)	49	0	4K	426	37.5%
SELP (OOP)	74	0	4K	348	37.5%
Shuf (IP)	49	0	0	426	50%

Table 1: Kernel Characteristics.
[1]Local Memory. [2]Shared Memory. [3]Standard lines of code. [4]Occupancy.

Shared memory is allocated, but is not used, to fix the occupancy level.

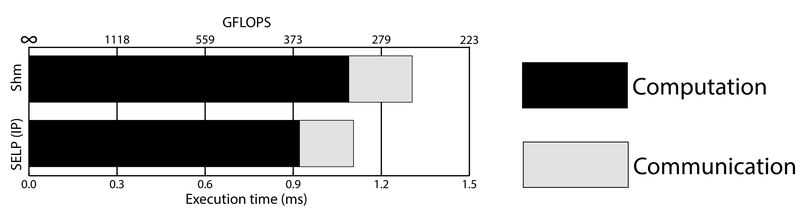
6. Insights & Conclusions

Insights

- Be wary of CUDA local memory allocation and usage. The CUDA compiler will allocate local memory if it cannot determine register indices at compile time. Accesses to local memory is slower than registers.

Conclusions

- SELP OOP improves the communication stage of FFT (gray bar) by 44%.
  - Overall speedup[1]: 1.08x Maximum speedup[3]: 1.19x
- Since shared memory is eliminated, SELP IP is able to run at a higher occupancy level. We indirectly accelerated the computation portion of FFT in addition to communication.
  - Overall speedup[2]: 1.17x Maximum speedup[3]: 1.19x



[1]SELP OOP @ 37.5% occupancy. [2]SELP IP @ 50% occupancy. [2]Shm @ 37.5% occupancy.