CSE 591: GPU Programming

Optimizing Your Application: Memory

Klaus Mueller

Computer Science Department

Stony Brook University

Code examples from Shane Cook "CUDA Programming"

Memory Transfer Basics

Fetch (transaction) size

- cached access introduced with Fermi
- in cache lines of 32 or 128 bytes
- 128 bytes is the default (a warp with one float for each thread)
- 64-byte fetches are not supported

Memory requests are fed into a queue

- individually serviced by the memory subsystem

To get maximum (peak) bandwidth per thread

- read 4 floats all at once as opposed to 4 individual reads
- use `float2/int2` or `float4/int4` vector types
- prefer a few large transactions over many small ones
- recall padding

Source of Limit (1)

Two types of kernel limitations

- memory latency/bandwidth
- instruction latency/bandwidth
- make sure you optimize for the one that needs it

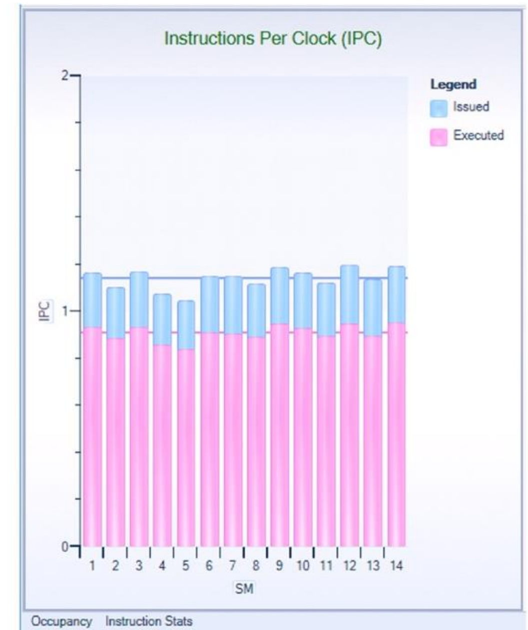
How to find what it is:

- comment out all arithmetic instructions and replace them with straight assignments to the result
- arithmetic instructions are calculations, branches, loops, …
- simply map input to output
- make sure to include all input data into the output or the compiler will remove the (apparently) extraneous reads
- re-time and calculate $100 \cdot (T_{before} - T_{now}) / T_{before}$
- if the percentage is very high then you are compute bound, else memory bound

Source of Limit (2)

Next

- with arithmetic still commented out look at the Analysis function and profile setting of Parallel Nsight
- blue bars mean poor coalescing
- serialization of global memory reads



Memory bound, what now?

- threads should have a column-based access pattern, not row-based
- access should be orderly and regular
- if it cannot be achieved consider pre-load into shared memory
- read data at begin of kernel and not right when needed
- this will increase register use – check limitations by monitoring number of scheduled warps (look for sudden drop-offs)

Source of Limit (3)

Compute bound, what now?

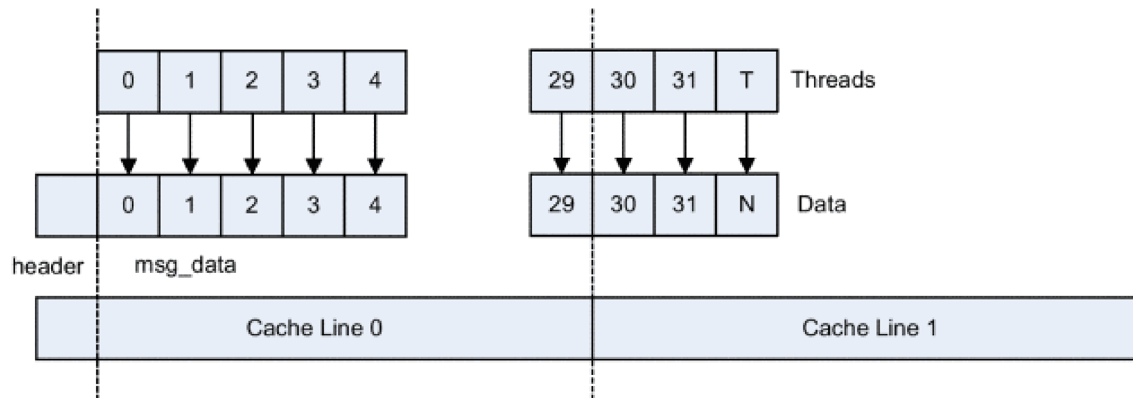
- check PTX code
- replace array indexes by pointer-based code
- replace slow multiples by faster adds
- replace multiplications/divisions by order of two by faster bit shifts
- move constant operations in loops outside
- unroll loops – but optimize for unroll factor
- replace double-precision with single-precision floats (or even half floats) if it can be justified
- watch out for floating point constants without the `F` postfix – will be doubles
- try the `–use_fast_math` compiler switch (faster 24 bit arithmetic)
- compile into ‘release’ mode – can give to 15% speedup and more

Memory Alignment

Another example:

```
#define MSG_SIZE 4096
typedef struct
{
 u16 header;
 u32 msg_data[MSG_SIZE];
} MY_STRUCT_T;
```

- 2-byte header produces offset for warp
- thread 30 and 31 cannot be served in a single fetch
- will incur a second 128-byte read
- applies to all subsequent warps



- solution: could move `header` to the end of the structure

Memory Alignment

Use padding to align to 32 byte boundaries

- use `cudaMallocPitch` when bins are irregularly sized – for example for prefix sums
- fixed size bins can often be handled by the programmer

Pad with a number that does not affect the result

- for a `min` operation use 0xFFFFFFFF
- for a `max` operation use 0
- other operations also always have such a benign value

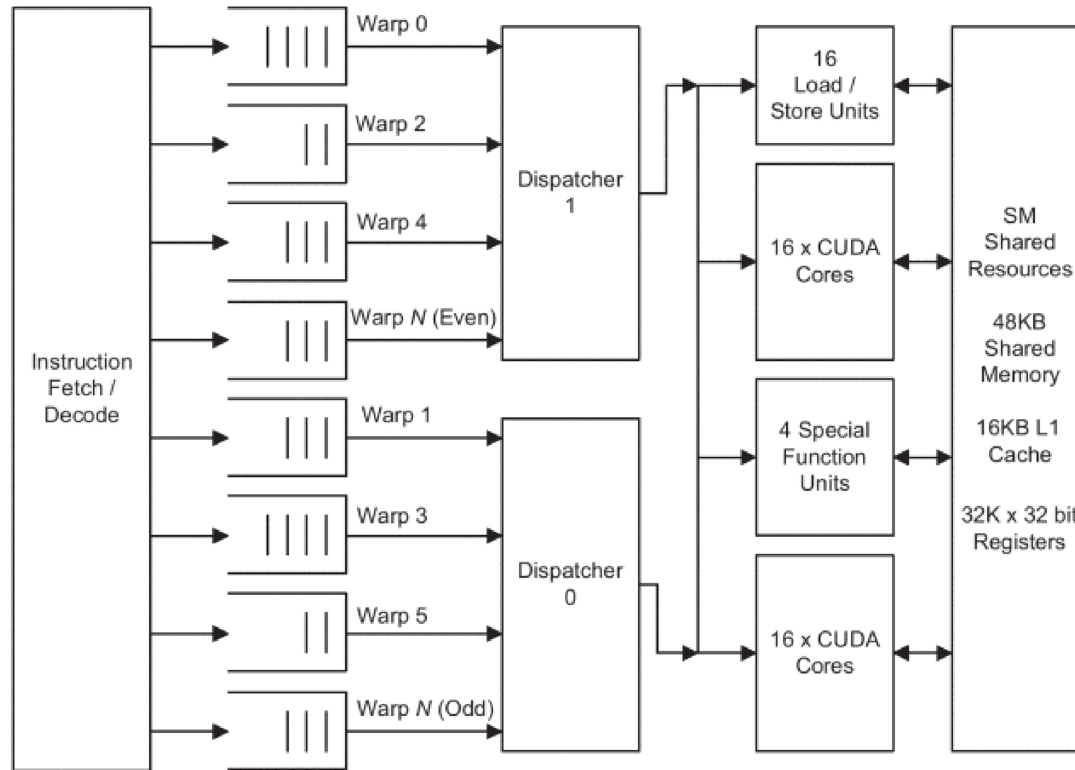
Memory Access to Compute Ratio

Ratio of memory operations to arithmetic operations

- want a ratio 1:10
- for every (global) memory fetch have 10 or more instructions
- this can include array index, loop calcs, branches, conditional …

CUDA Warp Dispatching

GF 100 (compute 2.0)



- per cycle two (4 in GF 104) instructions are issued (one per dispatcher)
- so need 2 independent warps to be present at minimum (4 for GF 104)
- hence we need 64 threads minimum per SM
- having less leaves one ore more dispatch unit idle
- also need a multiple of 32 threads to avoid idle state

Latency Hiding

Shared resources

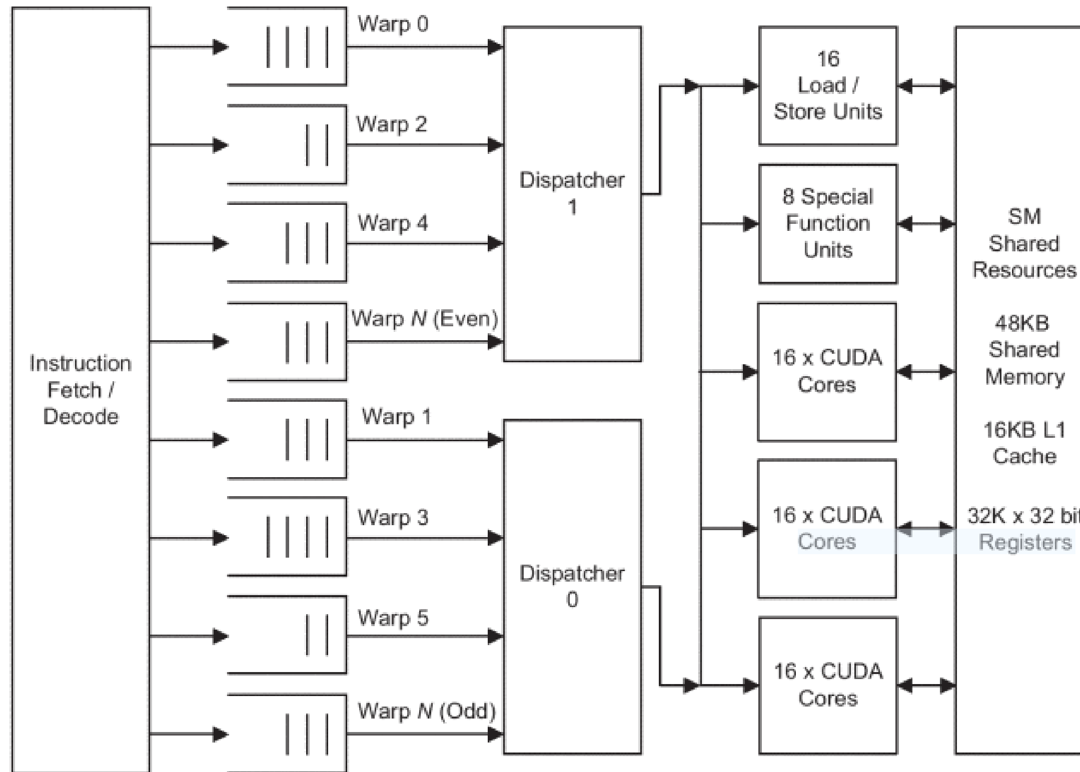
- need to have mix of (independent) instructions that utilize units well
- both CUDA cores and LSU are pipelined but only 16 units wide
- so a warp to either unit will need two cycles to complete
- there are four receivers for a dispatch: LSU, CUDA, SFU, CUDA
- but we have only two suppliers

Contingencies

- only a single warp can use the LSU and it takes two clock cycles
- only a single warp can use the SFU but it takes 8 clock cycles because there are only 4 units

CUDA Warp Dispatching

GF 104 (compute 2.1)

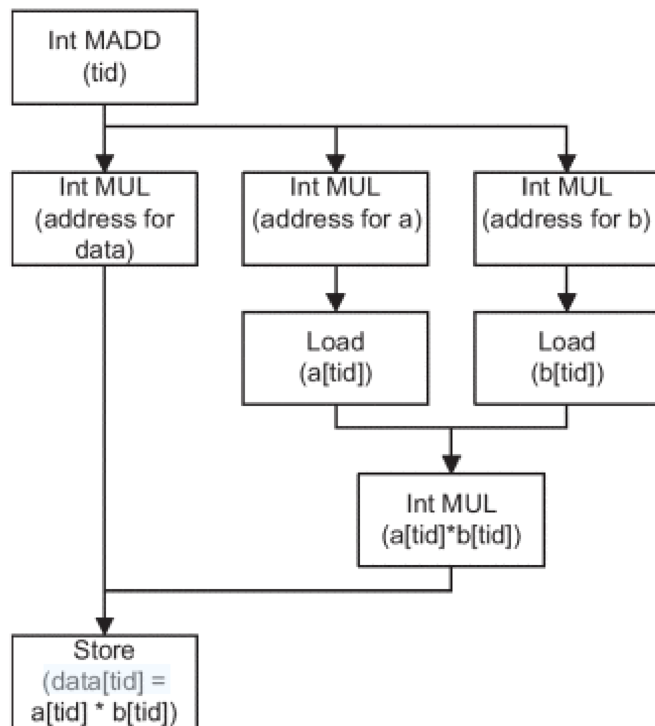


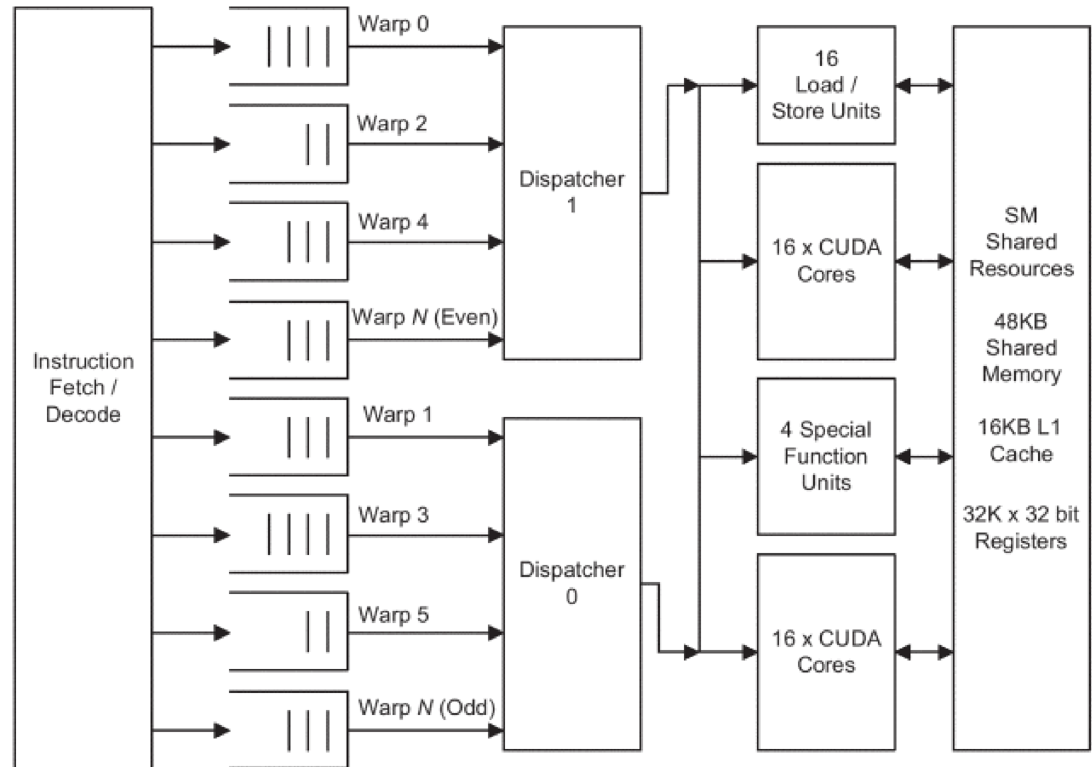
- better bandwidth than GF 100
- Kepler has 96 CUDA cores, and then puts two of these within an SM Thus there are four warp schedulers, eight dispatch units, two LSUs and two SFUs per SM.

Resource Utilization

Code example

```
int tid = (blockIdx.x * blockDim.x) + threadIdx.x;
data[tid] = a[tid] * b[tid];
```



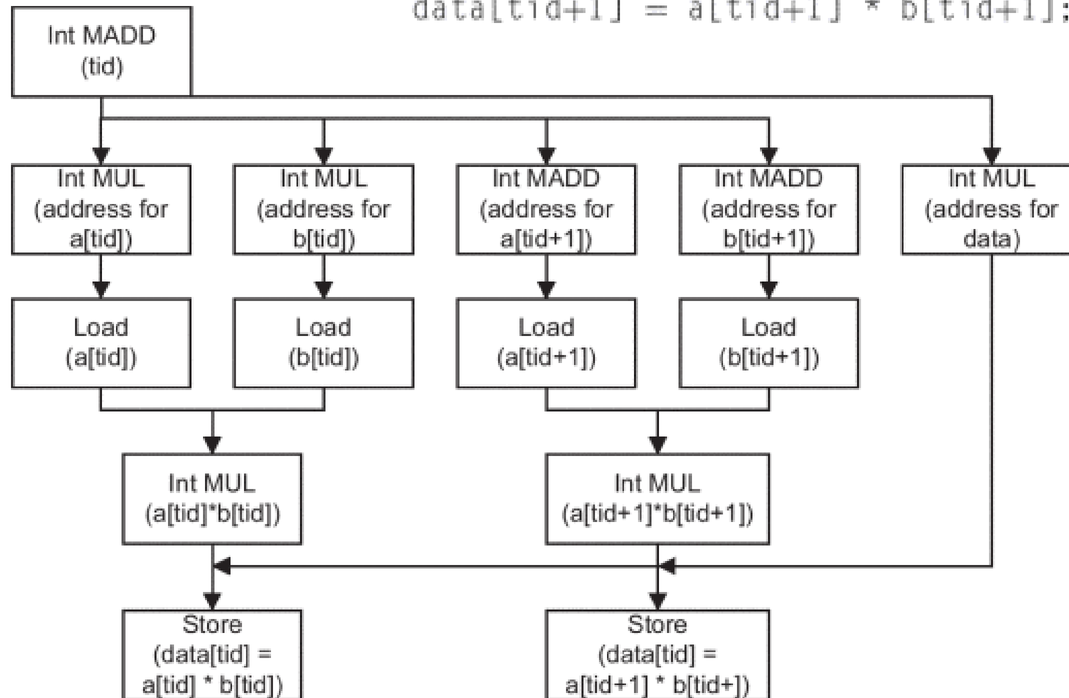


- instructions are dependent – each instruction blocks the following
- use of resources is essentially serialized

Better Resource Utilization

Loop unrolling

```
int tid = (blockIdx.x * blockDim.x) + threadIdx.x;
data[tid] = a[tid] * b[tid];
data[tid+1] = a[tid+1] * b[tid+1];
```



- introduces a second independent instruction stream
- now arithmetic operations overlap with load operations
- this makes better use of the resources
- shows the need for more larger instruction stream (arithmetic intensity)

Better Resource Utilization (2)

Remove possible dependencies

- when data and a, b space overlap

```
int tid = (blockIdx.x * blockDim.x) + threadIdx.x;
int a_0 = a[tid];
int b_0 = b[tid];
int a_1 = a[tid+1];
int b_1 = b[tid+1];
data[tid] = a_0 * b_0;
data[tid+1] = a_1 * b_1;
```

or

```
int tid = (blockIdx.x * blockDim.x) + threadIdx.x;
int2 a_vect = a[tid];
int2 b_vect = b[tid];
data[tid] = a_vect * b_vect;
```

- ‘vector’ loads and saves two 64-bit operands and is more efficient
- for vectors use `int2, int3, int4, float2, float3, float4`

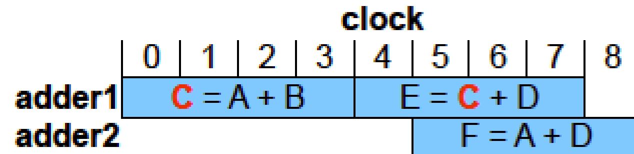
Instruction Level Parallelism (ILP): Another Example

Dependencies not permitting ILP (9 clock cycles)

C = A + B

E = C + D

F = A + D

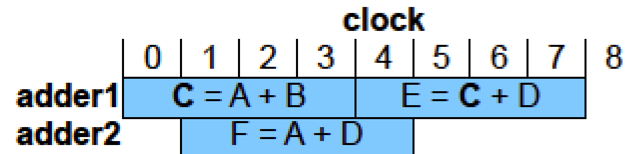


Instruction reordering for better ILP (8 clock cycles)

C = A + B

F = A + D

E = C + D



Computation and Load/Store: No ILP

```python
@cuda.jit('float32(float32, float32)', device=True)
def core(a, b):
    return a + b
```

Python

```python
@cuda.jit('void(float32[:], float32[:], float32[:])')
def vec_add(a, b, c):
    i = cuda.grid(1)
    c[i] = core(a[i], b[i])
```

Python

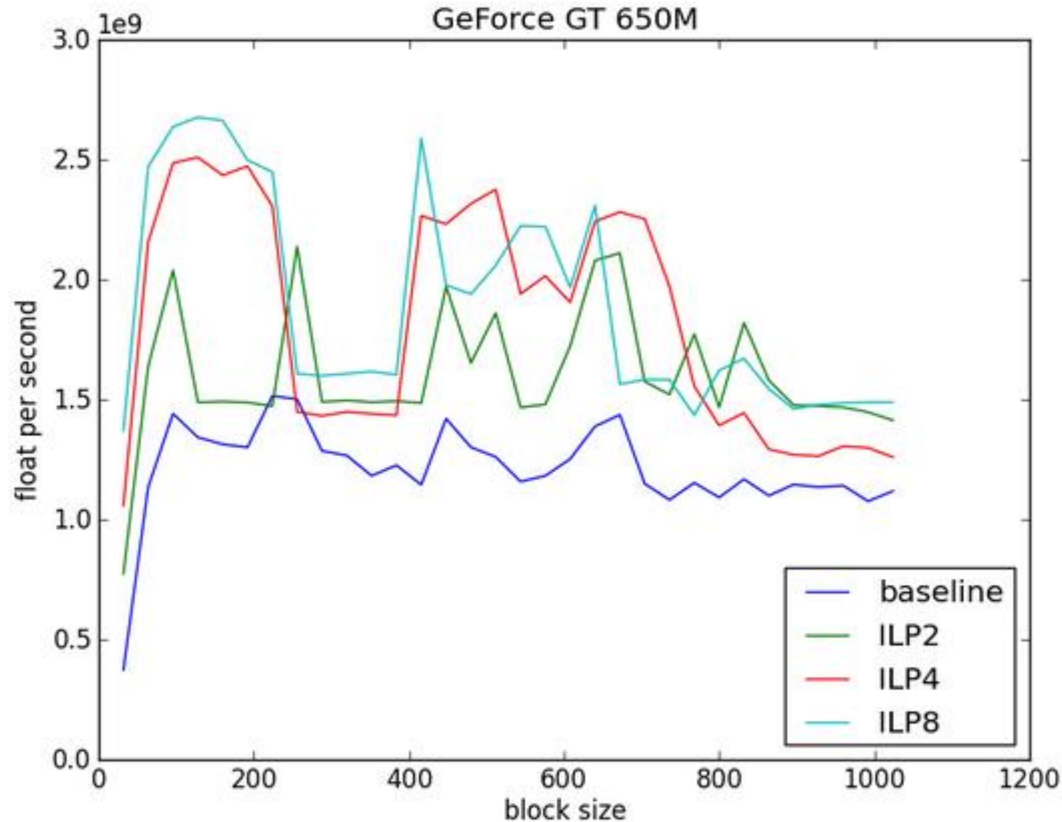
Computation and Load/Store: With ILP

```python
@cuda.jit('void(float32[:], float32[:], float32[:])')
def vec_add_ilp_x2(a, b, c):
    # read
    i = cuda.grid(1)
    ai = a[i]
    bi = b[i]

    bw = cuda.blockDim.x
    gw = cuda.gridDim.x
    stride = gw * bw

    j = i + stride
    aj = a[j]
    bj = b[j]

    # compute
    ci = core(ai, bi)
    cj = core(aj, bj)

    # write
    c[i] = ci
    c[j] = cj
```

Python

Computation and Load/Store: With ILP – Results

See http://continuum.io/blog/cudapy_ilp_opt for ILP=4 and 8



Other Considerations

Already discussed earlier (read more on book):

Loop fusion

- reduces redundant arithmetic
- and others

Kernel fusion

- avoids redundant memory transfers
- data just remains in shared memory (or caches)
- encourages data reuse

Shared memory

- can regularize irregular access patters to global memory