

Memory optimisations

CUDA Course
István Reguly

Outline

- Approaches to optimisation
- How the hardware does it
- Loads in Flight
- Iterative optimisation of a transpose example
 - occupancy
 - coalescing
 - shared memory
 - memory level parallelism

Approaches to optimisation

- Optimising the algorithm: to reduce the amount of work done to compute the answer
 - Or to improve access patterns, expose more parallelism, etc...
 - Most interesting kind of optimisation...
- Execution optimisation
 - Maximising the utilisation of computational resources, given the algorithm we have

Algorithm optimisation

- Reducing complexity ($O(..)$)
 - Always good to take a step back and think about the algorithm
 - First thing to do is figure out on paper which is going to be better
- Cons:
 - Ignores implementation details
 - Constants in big O notation are tricky...
 - Data locality!

Example: matrix operations

	MATRIX-VECTOR	MATRIX-MATRIX
Flop count	$3N^2 - N$	$3N^3 - N^2$
$O(\cdot)$ runtime	$O(N^2)$	$O(N^3)$
Memory ops	$2N^2 + N$	$3N^2$
Op per Access	1	$O(N)$

- Compare matrix-vector (GEMV) and matrix-matrix multiplication (GEMM)

Peak ratios

	PEAK GFLOP/S	PEAK GB/S	OPS/BYTE	OPS/WORD
K80 single	4368	240	~18	~72
K80 double	1456	240	~6	~48
Xeon 2690 single	416	68	~6*	~24*
Xeon 2690 double	208	68	~3*	~24*

- Without a lot of operations per data element, it's going to be bound by bandwidth!

*ignoring caches...

Memory operations

- To have computations and communications in balance, one needs a huge amount of ops/word
 - Rarely the case...
- We need to saturate the device bandwidth to get the best performance
 - Maximise the number of loads in flight

Loads in flight

- One memory load is 32 or 128 bytes (depends on cache config, texture loads are always 32)
- The latency of one transaction is ~400-600 clock cycles
- We need a whole lot of independent loads in flight: $288 \times 1024^3 / 32$ per second...

Matrix transpose

- NxN matrix, rows \leftrightarrow columns
- Algorithmic analysis:
 - $O(N^2)$
 - No computations



Example in SDK samples: 6_Advanced/transpose
Or transpose.cu on the website

Naive transpose

```
void reference_transpose(int rows, int cols, float *in, float *out) {  
    for (int i = 0; i < rows; ++i) {  
        for (int j = 0; j < cols; ++j) {  
            out[i*rows+j] = in[j*cols+i];  
        }  
    }  
}
```

i loop's iterations are independent!
Why not parallelise it on the GPU?

Naive CUDA implementation

```
__global__ void transpose1(int rows, int cols, float *in, float *out) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    for (int j = 0; j < cols; ++j) {  
        out[i*rows+j] = in[j*cols+i];  
    }  
}
```

Launch 1D grid of 1D blocks, with 256 threads/block
Matrix size $2048^3 \rightarrow 8$ blocks

Runs in 2.1ms on a K80. Is that good?

Performance

KERNEL	FLOAT	DOUBLE
transpose1	15.25 GB/s	29.61 GB/s

Tesla K80

Telling how close we are

- When we know the theoretical bounds:
 - How many loads would be required by the algorithm
 - How many operations would be required by the algorithm
 - bytes/sec or flops/s
- Compare to theoretical

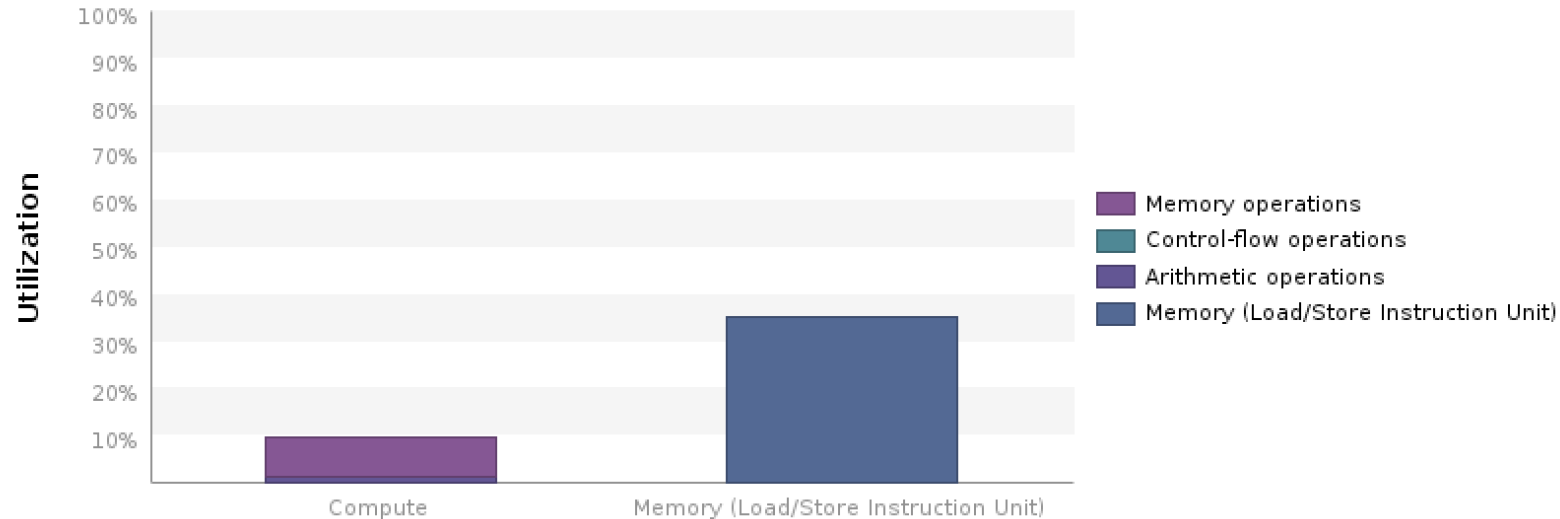
Get info out of the profiler

- Gives metrics for throughput from L2 and DRAM
 - dram_read_throughput
 - l2_l1_read_throughput
 - l2_tex_read_throughput
- These provide numbers from the L2/DRAM perspective
 - Includes cache misses, that could have been hits
 - Includes bandwidth used moving bytes not used by the algorithm
 - Includes ECC
- Not really indicative of real performance...

Profiling naive version

i Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "Tesla K80". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



Latency analysis

- We have 8 blocks of 256 threads, each doing only one load or store at the same time
- But we have 13 SMX units available
- Far too few loads in flight!

Grid Size Too Small To Hide Compute And Memory Latency

The kernel does not execute enough blocks to hide memory and operation latency. Typically the kernel grid size must be large enough to fill the GPU with multiple "waves" of blocks. Based on theoretical occupancy, device "Tesla K80" can simultaneously execute 8 blocks on each of the 13 SMs, so the kernel may need to execute a multiple of 104 blocks to hide the compute and memory latency. If the kernel is executing concurrently with other kernels then fewer blocks will be required because the kernel is sharing the SMs with those kernels.

Optimization: Increase the number of blocks executed by the kernel.

[More...](#)

- We have 8 blocks of 256 threads each
- But we have 13 SMX units
- Far too few loads in flight

transpose1(int, int, float*, float*)

Start	319.104 ms (319,104, ...)
End	321.101 ms (321,100, ...)
Duration	1.997 ms (1,996,711 r ...)
Grid Size	[8,1,1]
Block Size	[256,1,1]
Registers/Thread	17
Shared Memory/Block	0 B
Occupancy	
Achieved	⚠ 12.5%
Theoretical	100%

Results

⚠ **Grid Size Too Small To Hide**

The kernel does not execute enough to fill the GPU with multiple threads. It can only execute 8 blocks on each of the 15 SMs, which is not enough to hide the memory latency. If the kernel is executed many times, the SMs will be sharing the SMs with those kernels.

Optimization: Increase the number of threads per block.

Shared Memory Config	
Shared Memory Required	112 KiB
Shared Memory Executed	112 KiB
Shared Memory Bank	4 B

store at the same time

al grid size must be large
<40c" can simultaneously
to hide the compute and
quired because the kernel is

[More...](#)

Occupancy and bandwidth

- The memory controllers can handle X number of transactions per second
- Golden rule: always keep the “bottleneck” resource busy
 - Should be purely bandwidth bound
- More independent pieces of work, more threads, more parallelism, more simultaneous loads -> more loads in flight
- GPU is good at throughput, not latency (unlike CPUs)
 - Only way to hide latency is through parallelism - some threads wait, others have work to do
 - ->Occupancy is the only way to hide latency (well...)

Find more parallelism

- Profiler says we need a bigger computational grid
 - so we need more parallelism
- We need to find more parallelism
- Have each thread do less, distribute it across more threads
 - Let's parallelise both loops and use a 2D grid!

2D transpose

```
__global__ void transpose2(int rows, int cols, float *in, float *out) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    int j = blockIdx.y * blockDim.y + threadIdx.y;  
  
    out[i*rows+j] = in[j*cols+i];  
}
```

- Launch a 2D grid, with 2048 total threads in each dimension

Performance

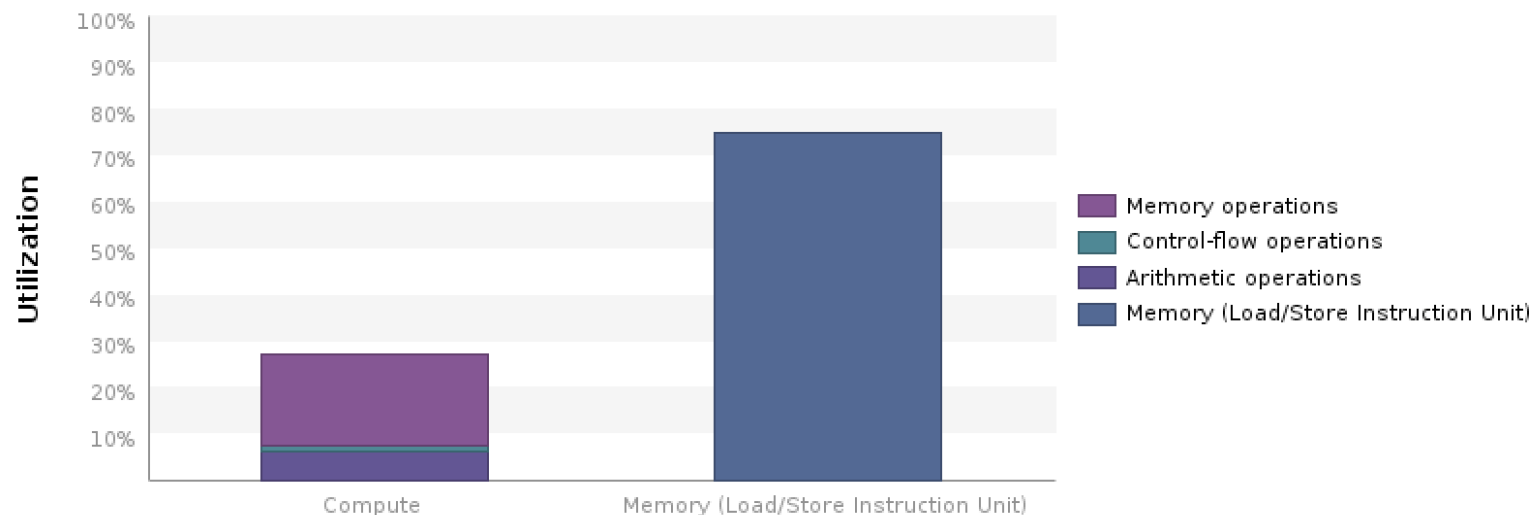
KERNEL	FLOAT	DOUBLE
transpose1	16.96 GB/s	33.94 GB/s
transpose2 (2D)	59.8 GB/s	109.5 GB/s
Improvement	3.5x	3.2x

Tesla K80

Profiling 2D version

i Kernel Performance Is Bound By Memory Bandwidth

For device "Tesla K80" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the load/store instruction units within the multiprocessors.



Bandwidth analysis

⚠ Global Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern.

Optimization: Select each entry below to open the source code to a global load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access.

[More.](#)

▼ Line / File	transpose.cu - /home/ireguly/projects/cuda_course
24	Global Store L2 Transactions/Access = 32, Ideal Transactions/Access = 4 [4194304 L2 transactions for 131072 total executions]

Line 24: `out[i*rows+j] = in[j*cols+i];`

Only complains about store access pattern!

Mapping programming model to hardware

- You launch a thread block, with each thread executing the same code
- Each block gets assigned to an SM
- An SM has 192/128 little CUDA cores - but these are not independent
- Threads are bundled together into groups of 32 called warps
 - Threads in a 2D thread block are mapped to warps in a row-contiguous way
 - All threads in a warp are executing the same instruction, just with different data (lockstep)

Warp execution

- 1 program counter per warp (there are four on an SMM/SMX)
- The next instruction is issued to 32 parallel CUDA cores
 - Each has its own set of registers
 - So same instruction, just with different operands
 - **Note: even the register indexes are the same for a given instruction, just their contents are different**
- Repeat until done

Executing a load instruction

- When encountering a load instructions, all 32 threads will load from individual addresses in memory
- LD.E R2,[R6]

THR	0	1	2	3	4	5	6	...
R6	0x0FD3A0	0x0FD3A4	0x0FD3A8	0x0FD3AB	0x0FD3B0	0x0FD3B4	0x0FD3B8	

Executing a load instruction

- From the point of view of the thread
 - Asks for a small chunk of memory (4-8 bytes)
 - $f = \text{array}[\text{offset} + \text{threadIdx.x}]$
 - Called a **request** or **access**
- From the SM's perspective
 - **Issues** a load for the cache line that contains $\text{array}[\text{offset} + \text{threadIdx.x}]$
 - May need to issue multiple loads to satisfy all requests from all threads in the warp
 - After the first, each further cache line load is a **replay**
- From the L2/DRAM perspective
 - If the segment is valid in L2, return it from there
 - Otherwise fill L2 from DRAM and send to value to SM

Views of bandwidth

- Thread's view
 - How much data the thread receives per unit time
 - Usually very low number - but there are a lot of threads!
- SM's view
 - How much data is delivered to all the threads active on the SM
 - Receives cache lines
- L2/DRAM view
 - How much data is moving between DRAM and L2 cache

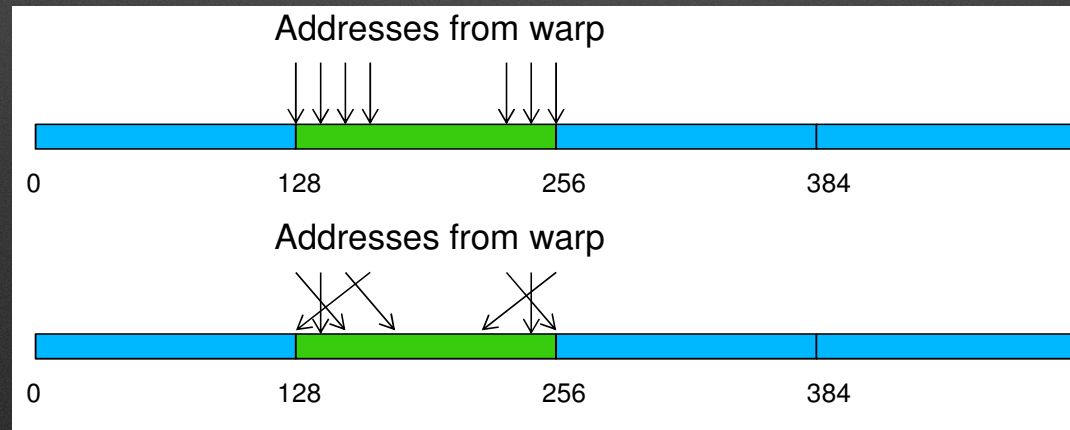
Aggregate bandwidth from thread perspective
is not the as from L2/DRAM perspective!

Memory transactions and coalescing

- Access to global memory triggers transactions
 - size of 32 or 128 bytes
 - aligned to line length
 - always fully R/W
- Degree of coalescing: $\frac{\text{\#of bytes requested}}{\text{\#of bytes used}}$
 - more memory read than used -> performance penalty

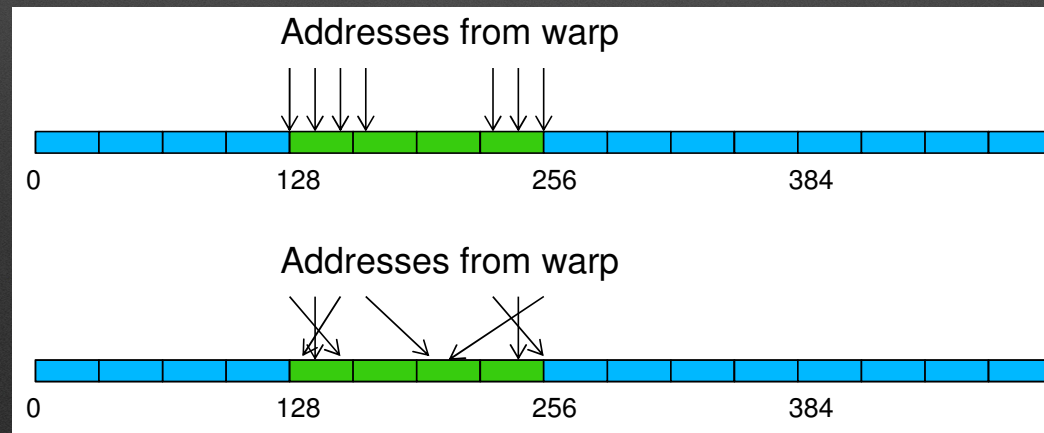
L1-Cached Thread Index Access

- 32 adjacent threads requesting 32 adjacent words
 - aligned but may be permuted
 - all fall in one cache line (128B)
 - 1 transaction



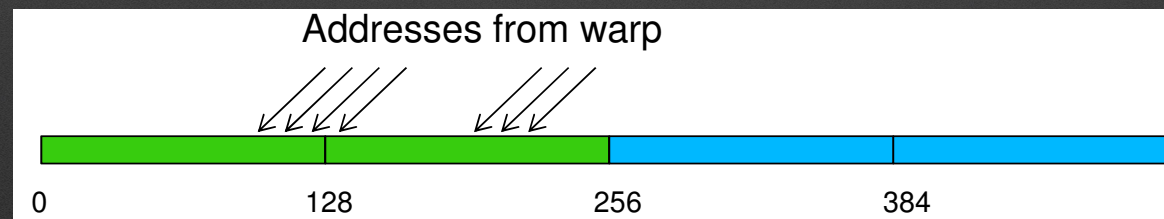
L2-Cached Thread Index Access

- 32 adjacent threads requesting 32 adjacent words
 - all addresses fall within 4 segments
 - 4 transactions



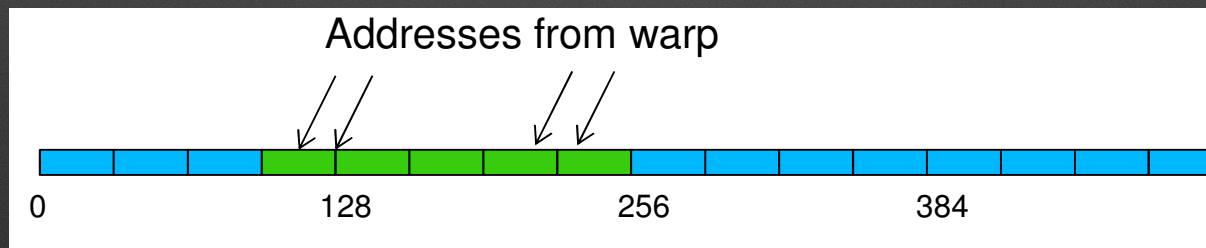
L1-Cache Shifted Access

- 32 adjacent threads requesting 32 adjacent words
 - misaligned
 - 2 transactions
 - Cache line utilisation 50%
- 2D stencil operations
- In a 2D setting where leading dimension is not a multiple of cache line length - use padding



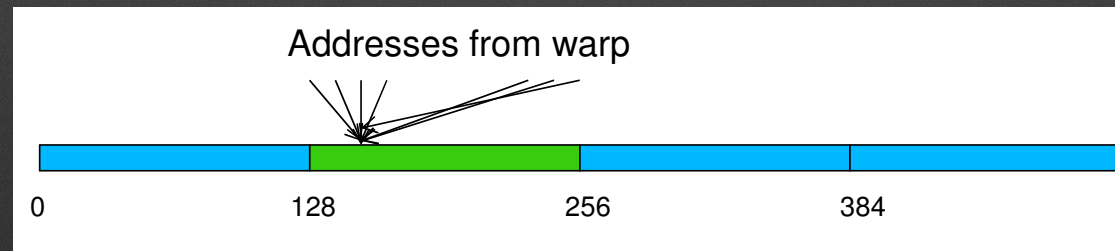
L2-Cached Shifter Access

- 32 adjacent threads requesting 32 adjacent words
 - all addresses fall within 5 segments
 - 5 transactions - 80% utilisation



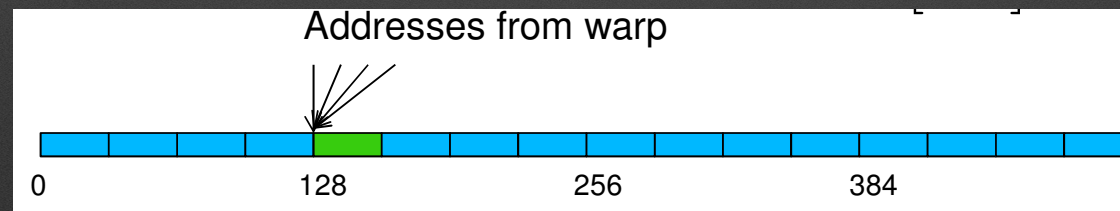
L1-Cached Single Access

- All 32 threads access the same word in memory
- Full 128B cache line is transferred - 3.125% utilisation



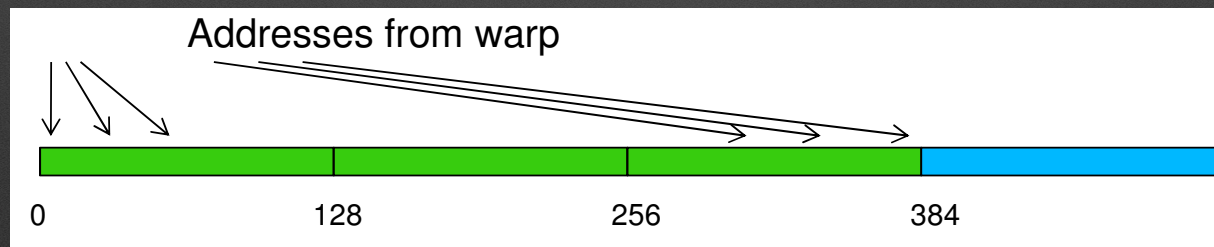
L2-Cached Single Access

- All 32 threads access the same word in memory - same segment
- Full 32B cache line is transferred - 12.5% utilisation



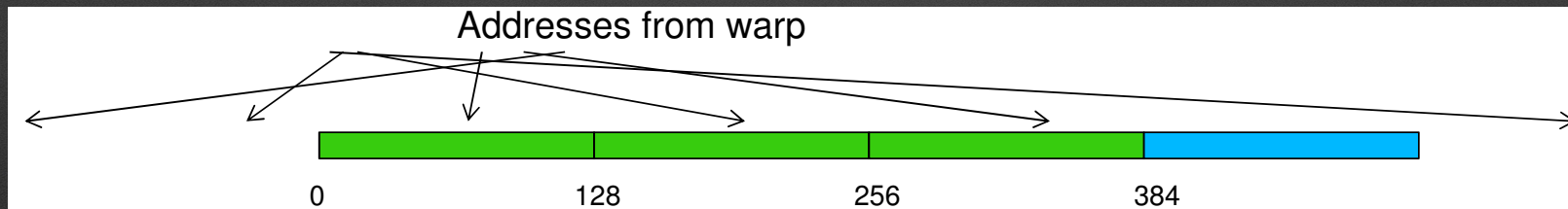
L1-Cache Strided Access

- 32 adjacent threads requesting 32 words with stride 3
 - addresses fall in 3 cache lines
 - 3 transactions
 - Cache line utilisation 33%
- Typical of 3D coordinate or RGB accesses -> use SoA layout!



L1-Cache Fully Random Access

- 32 adjacent threads requesting 32 words with random addresses
 - addresses fall in 32 different cache lines
 - 32 transactions
 - Cache line utilisation 3.125%
- Pointer chasing, trees, etc.



Coalescing

- The way to minimise the number of operations required to satisfy all requests from a single warp's one load instruction
- If multiple addresses are in the same cache line, it only gets moved once
 - The more in the same cache line the better
 - Best: 32 addresses, each 4 bytes - a single 128B load
 - One L1-L2 load and 4 L2-DRAM loads
 - Worst: separate transactions for each - 31 replays
 - 32 L1-L2 loads and 128 L2-DRAM loads

Good access patterns

- On CPUs, the advice is to do stride-1 accesses

```
for (int i = 0; i < n; ++i) {  
    x += data[i];  
}
```

- Increases cache hits, easy to compute, prefetching work well, etc...

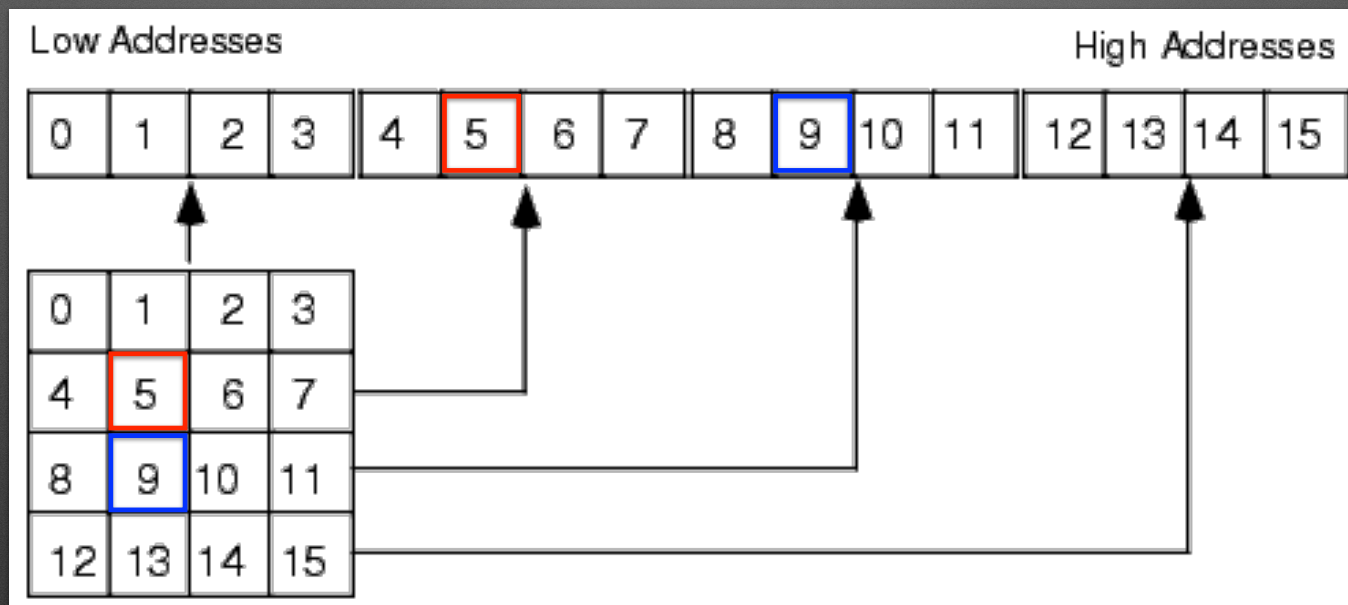
Good access patterns

- Basically “transposed” to threads in GPU code:

```
for (int i = threadIdx.x; i < n; i += blockDim.x) {  
    x += data[i];  
}
```

- Same idea of accessing the same cache line, except we do 32 accesses at the same time
- You need locality across threads for one instruction instead of locality across subsequent instructions for one thread

Back to transpose



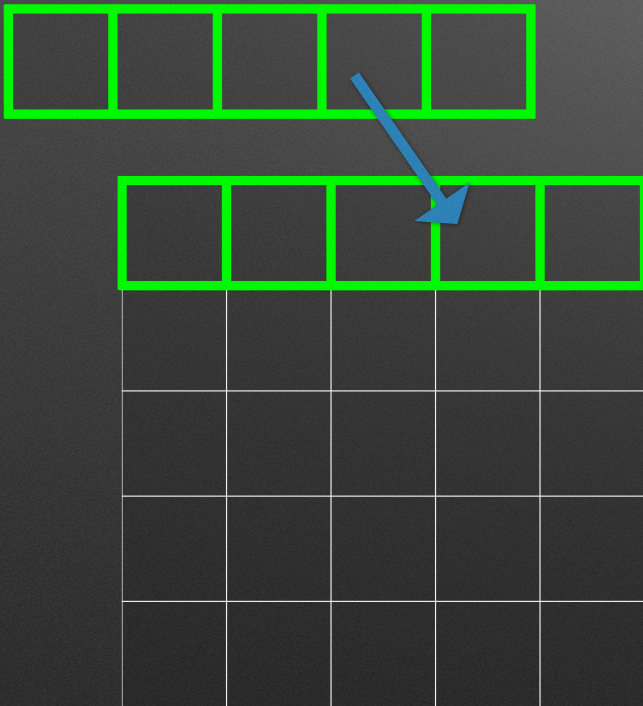
Stride-1 column accesses are stride-N accesses!

Back to transpose

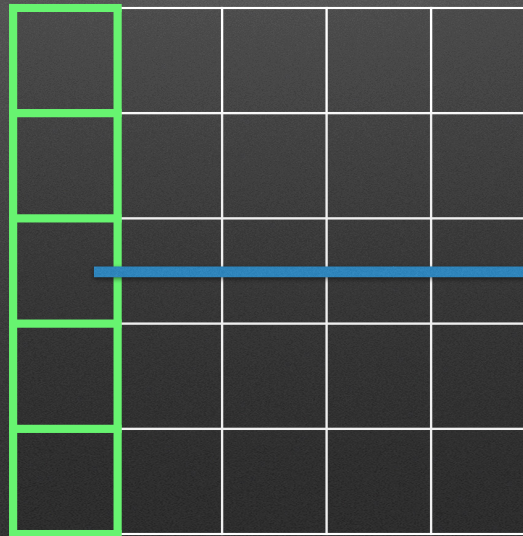
- So what can we do with our transpose code?
 - Reads are fine, but writes are bad
- Row read and row write is the most effective - how could we achieve that?
- Transpose in shared memory

Shared memory transpose

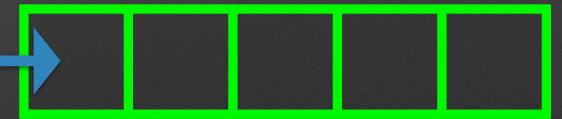
Warp reads row
to shared mem



Warp reads column
from shared mem



Warp stores
contiguous row



Shared memory transpose

```
#define TILE_SIZE 16
__global__ void transpose3(int rows, int cols, float *in, float *out) {
    __shared__ float tile[TILE_SIZE][TILE_SIZE];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    tile[threadIdx.y][threadIdx.x] = in[j*cols+i];

    __syncthreads();

    i = blockIdx.y * blockDim.y + threadIdx.x;
    j = blockIdx.x * blockDim.x + threadIdx.y;
    out[j*rows+i] = tile[threadIdx.x][threadIdx.y];

}
```


Syncthreads

- Producer-consumer thinking
 - The write consumes the read value
 - With previous examples, it was the same thread
 - With shared memory transpose, one thread produces the value and another consumes it
- We have to make sure it was produced before we try to consume it
 - Great tool if you are running into problems: `cuda-memcheck --tool racecheck`

Performance

KERNEL	FLOAT	DOUBLE
transpose1	16.96 GB/s	33.94 GB/s
transpose2 (2D)	59.8 GB/s	109.5 GB/s
transpose3 (coalesced)	80.54 GB/s	102.3 GB/s
Improvement	1.32x	0.93x

Tesla K80

Bandwidth analysis

⚠ Shared Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.

Optimization: Select each entry below to open the source code to a shared load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access.

▼ Line / File	transpose.cu - /home/ireguly/projects/cuda_course
39	Shared Load Transactions/Access = 4, Ideal Transactions/Access = 1 [524288 transactions for 131072 total executions]

Line 39: `out[j*rows+i] = tile[threadIdx.x][threadIdx.y];`

Shared memory bank conflicts

- Shared memory is organised into banks
 - 32 banks, new bank every 4 bytes (or possibly 8 in Kepler)
 - $\text{Bank} = (\text{address}/4)\%32$
 - Can read one 32-bit word per bank per clock
 - Warp access: reads 32 words from shared memory per instruction

Shared memory bank conflicts

BANK 1	BANK 2	BANK 3	BANK 4
0	1	2	3
32	33	34	35
64	65	66	67
96	97	98	99

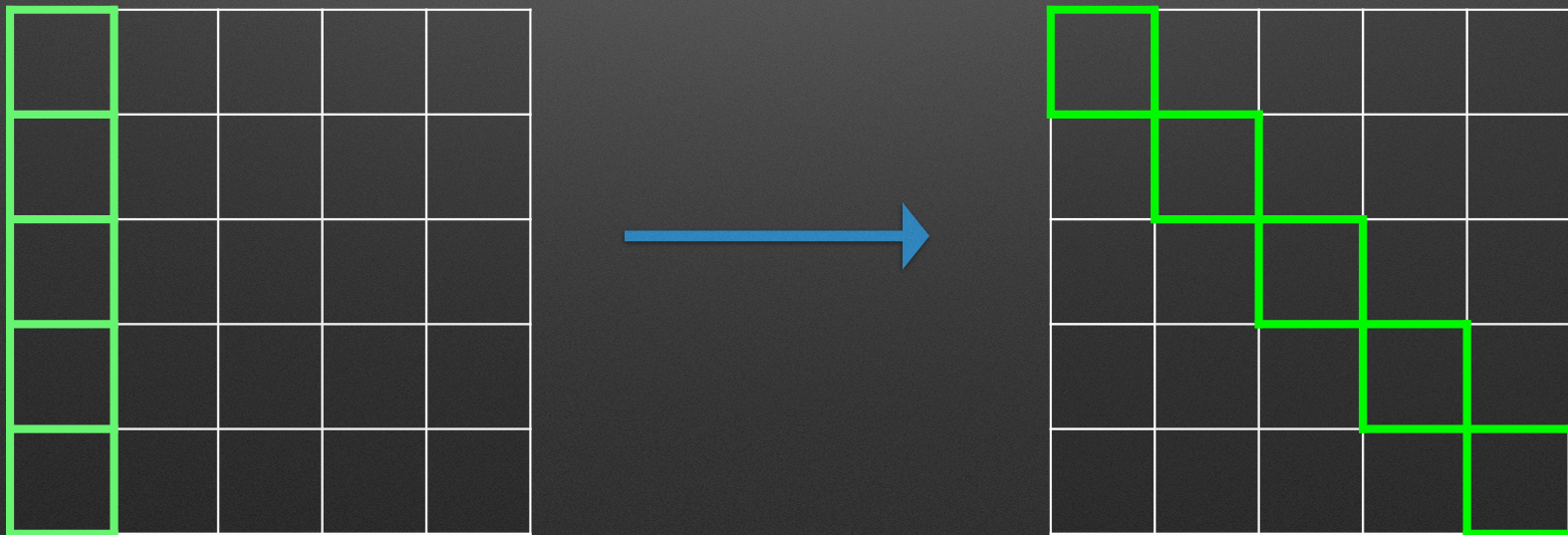
- What happens when we write
 - $\text{idx} = \text{threadIdx.y} * 32 + \text{threadIdx.x}$
 - adjacent banks...
- ...
- What happens when we read
 - $\text{idx} = \text{threadIdx.x} * 32 + \text{threadIdx.y}$
 - same bank...
- Requires replays - 32x the cost!

Impact of replays

- If a warp has to replay instructions, it cannot proceed until all replays are completed - a load may take up to 32x the number of instructions
- Occupies the warp scheduler - no useful operations in the meanwhile
- Decreases number of loads in flight...
- Note: we saw replays for both shared memory bank conflicts and non-coalesced global accesses - different reasons but the same effect

Avoiding bank conflicts

- Problem is with the index computation:
 - $\text{bank} = (\text{threadIdx.x} * 32 + \text{threadIdx.y}) \% 32$, where threadIdx.y is the same of all thread in a warp
- So let's pad our array to size 33:
 - $\text{bank} = (\text{threadIdx.x} * 33 + \text{threadIdx.y}) \% 32$



Padded shared memory

```
#define TILE_SIZE
__global__ void transpose3(int rows, int cols, float *in, float *out) {
    __shared__ float tile[TILE_SIZE][TILE_SIZE+1];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    tile[threadIdx.y][threadIdx.x] = in[j*cols+i];

    __syncthreads();

    i = blockIdx.y * blockDim.y + threadIdx.x;
    j = blockIdx.x * blockDim.x + threadIdx.y;
    out[j*rows+i] = tile[threadIdx.x][threadIdx.y];
}
```

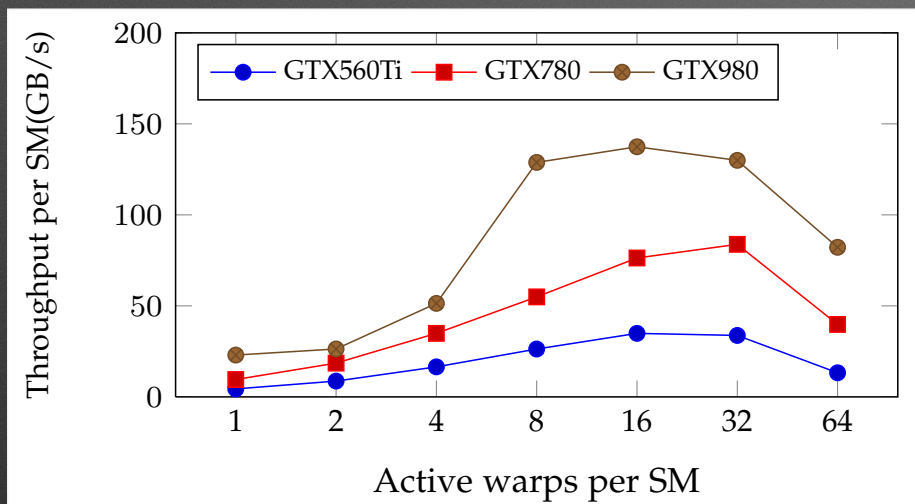

Performance

KERNEL	FLOAT	DOUBLE
transpose1	16.96 GB/s	33.94 GB/s
transpose2 (2D)	59.8 GB/s	109.5 GB/s
transpose3 (coalesced)	80.54 GB/s	102.3 GB/s
transpose4 (no bank conf)	127 GB/s	165.7 GB/s
Improvement	1.58x	1.61x

Note: on Maxwell, cost of bank conflicts is 2-5x less

Tesla K80

Shared memory performance



Shared Memory Access Latency with Bank Conflicts

Bank conflict	2-way	4-way	8-way	16-way	32-way
GTX980	30	34	42	58	90
GTX780	82	96	158	257	484
GTX560Ti	87	162	311	611	1209

XinXin Mei et. al. “Dissecting GPU Memory Hierarchy through Microbenchmarking”

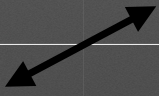
Why is double precision better?

- 127 GB/s vs. 165 GB/s
- Single precision:
 - 1 warp, 32 loads each, 4 bytes each, fully coalesced
 - 1 cache line; 4 DRAM transactions
- Double precision:
 - 1 warp, 32 loads each, 8 bytes each, fully coalesced
 - 2 cache lines; 8 DRAM transactions
- More loads in flight!

Even more loads in flight

- We can create a little loop that transposes multiple values per thread - more independent loads!
- Total number of loads is still the same, but so is the number of active threads!

KERNEL	FLOAT	DOUBLE
transpose 4 (1/thread)	127.8 GB/s	165.7 GB/s
transpose 5 (2/thread)	161.1 GB/s	173.1 GB/s



We can keep increasing it until we run into occupancy problems

Memory level parallelism

- Multiple independent memory load operations in one thread
 - Issue all loads before the value of any is consumed
 - Makes better use of loads in flight
- Related to Instruction Level Parallelism
 - Techniques for achieving it are quite similar
 - Requires independent operations within the thread
 - MLP usually creates some ILP

MLP & ILP

```
LD.E R0, [R8];  
IMUL.U32.U32 R6,R5,0x84;  
ISCADD R6,R3,R6,0x2;  
STS [R6], R0;  
BAR.SYNC 0x0;
```

Original

With ILP & MLP

```
LD.E R5, [R14];  
IMAD.HI.X R13,R8,0x4,R15;  
STS [R9+0x420],R7;  
IMAD R14.CC,R8,0x4,R12;  
LD.E R7,[R12];  
IMAD.HI.X R15,R8,0x4,R13  
STS [R9+0x630],R6;  
IMAD R10.CC,R8,0x4,R13;  
LD.E R6,[R14];  
IMAD.HI.X R11,R4,0x4,R15;  
STS [R9+0x840],R5;  
...  
BAR.SYNC 0x0;
```


Overall performance

KERNEL	FLOAT	DOUBLE
transpose1	16.96 GB/s	33.94 GB/s
transpose2 (2D)	59.8 GB/s	109.5 GB/s
transpose3 (coalesced)	80.54 GB/s	102.3 GB/s
transpose4 (no bank conf)	127 GB/s	165.7 GB/s
transpose5 (multiple elem)	161.1 GB/s	173.1 GB/s
Improvement	9.5x	5.24x

Summary

- Without large amount of data-reuse (ops/byte), codes will be bound by operand delivery
 - Bandwidth and/or Latency
- Bandwidth saturation requires many loads in flight
- Best practices for GPU memory utilisation
 - Address Coalescing: Efficient use of memory system
 - Use shared memory to restructure loads/stores into coalesced patterns
 - Latency hiding
 - Occupancy, Instruction level parallelism

Local memory

- Due to the way the hardware works, for a given instruction, you always need to use the same registers across all threads in a warp
 - Registers cannot be dynamically indexed
- If you have local arrays (per thread arrays)
 - Make sure their size is known at compile time
 - Make sure all threads access the same element of the array at the same time
 - Make sure the index is known at compile time: unroll loops!

Spilling

- If you use too big arrays, or indexing cannot be determined at compile-time, or you artificially restricted the number of registers:
 - The compiler will “spill” registers
 - Basically puts them in global memory in a way that will result in coalesced accesses
 - On Kepler, by default it is the only thing cached in L1
 - On Maxwell by default only cached in L2

Static indexing

```
__global__ void kernel1(float * buf)
{
    float a[2];
    ...
    float sum = a[0] + a[1];
    ...
}
```

```
__global__ void kernel2(float * buf)
{
    float a[5];
    ...
    float sum = 0.0f;
    #pragma unroll
    for(int i = 0; i < 5; ++i)
        sum += a[i];
    ...
}
```


Static indexing - unrolled

<code>float sum = 0.0F;</code>		8192	<code>LD.E R2, [R6+0xc];</code>
<code>#pragma unroll</code>		8192	<code>FADD.FTZ R4, R4, R0;</code>
<code>for(int i = 0; i < 5; ++i)</code>		8192	<code>LD.E R0, [R6+0x10];</code>
<code>sum += a[i];</code>		8192	<code>FADD.FTZ R3, R4, R3;</code>
		8192	<code>FADD.FTZ R3, R3, R2;</code>
<code>buf[tid] = sum;</code>		8192	<code>FADD.FTZ R0, R3, R0;</code>
		8192	<code>ST.E [R6], R0;</code>

Assembly shows, that it was unrolled
Uses a total of 4(!) operations

Dynamic indexing

```
__global__ void kernel3(float * buf, int start_index)
```

```
{
```

```
    float a[6];
```

```
    ...
```

```
    float sum = 0.0f;
```

```
    #pragma unroll
```

```
    for(int i = 0; i < 5; ++i)
```

```
        sum += a[start_index + i];
```

```
    ...
```

```
}
```

float sum = 0.0F;	8192	STL.64 [R1+0x10], R8;
#pragma unroll	8192	LDL R6, [R7];
for(int i = 0; i < 5; ++i)	8192	LDL R0, [R7+0x4];
sum += a[start_index + i];	8192	LDL R5, [R7+0x8];
	8192	FADD.FTZ R6, R6, R0;
	8192	LDL R4, [R7+0xc];
	8192	LDL R0, [R7+0x10];
	8192	FADD.FTZ R5, R6, R5;
buf[tid] = sum;	8192	FADD.FTZ R4, R5, R4;
	8192	FADD.FTZ R0, R4, R0;
	8192	ST.E [R2], R0;

But at least we get nice coalesced accesses...

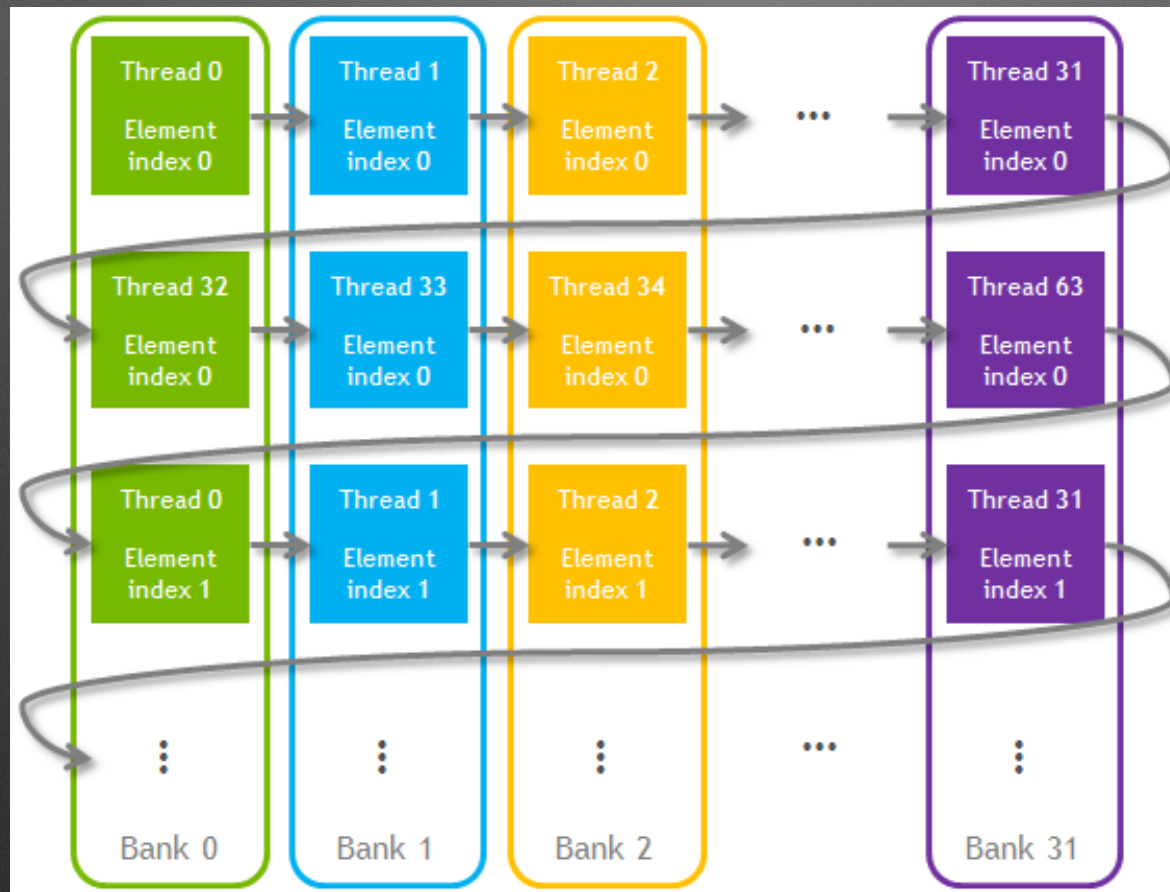
Dynamic, non-uniform indexing

```
#define ARRAY_SIZE 32
__global__ void kernel4(float * buf, int * indexbuf)
{
    float a[ARRAY_SIZE];
    ...
    int index = indexbuf[threadIdx.x + blockIdx.x * blockDim.x];
    float val = a[index];
    ...
}
```

How many replays?

Based on how many different indexes...

Put it in shared memory



One bank for
each thread

No bank conflicts
guaranteed!

Shared memory code

```
// Should be multiple of 32
#define THREADBLOCK_SIZE 64
// Could be any number, but the whole array should fit into shared memory
#define ARRAY_SIZE 32
```

```
__device__ __forceinline__ int no_bank_conflict_index(int thread_id,
                                                    int logical_index)
```

```
{
    return logical_index * THREADBLOCK_SIZE + thread_id;
}
```

```
__global__ void kernel5(float * buf, int * index_buf)
```

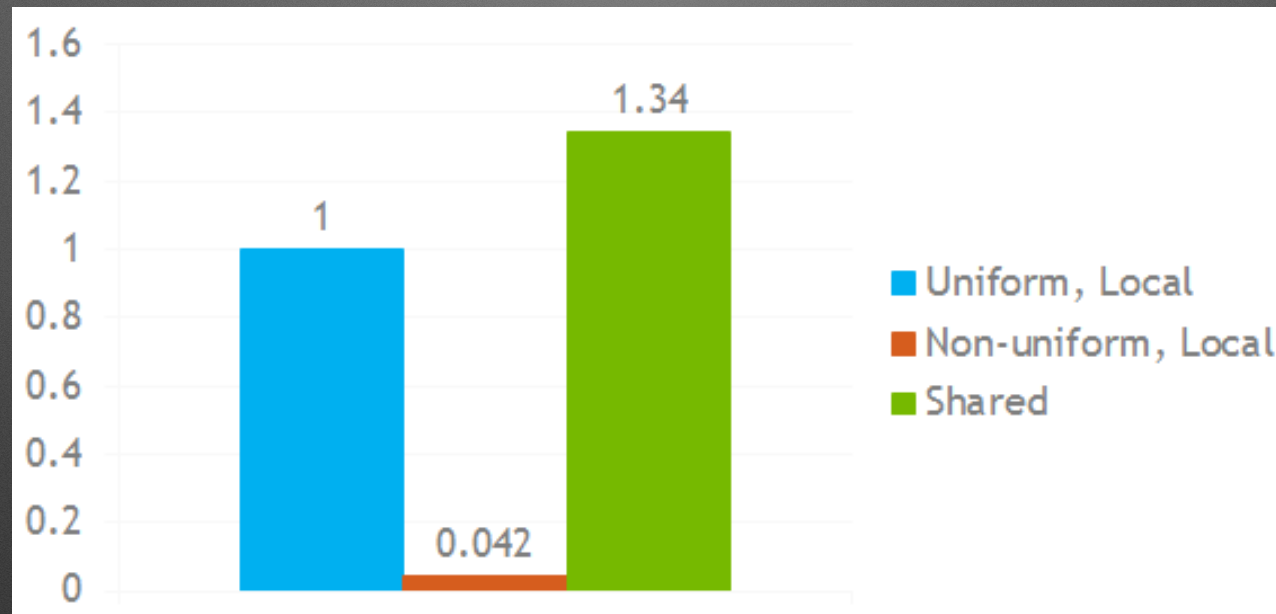
```
{
    // Declare shared memory array A which will hold virtual
    // private arrays of size ARRAY_SIZE elements for all
    // THREADBLOCK_SIZE threads of a threadblock
    __shared__ float A[ARRAY_SIZE * THREADBLOCK_SIZE];
    ...
    int index = index_buf[threadIdx.x + blockIdx.x * blockDim.x];

    // Here we assume thread block is 1D so threadIdx.x
    // enumerates all threads in the thread block
    float val = A[no_bank_conflict_index(threadIdx.x, index)];
    ...
}
```

```
float x = A[no_bank_conflict_index(threadIdx.x, index)];
float y = A[no_bank_conflict_index(threadIdx.x, index + 1)];
float z = A[no_bank_conflict_index(threadIdx.x, index + 2)];
float w = A[no_bank_conflict_index(threadIdx.x, index + 3)];
sum = x + y + z + w;
A[no_bank_conflict_index(threadIdx.x, index)] = sum;
```

```
STS [R9], R8;
SHF.L.W R10, RZ, 0x2
LDS R6, [R10+0x100];
LDS R7, [R10];
LDS R4, [R10+0x200];
FADD.FTZ R7, R7, R6;
LDS R3, [R10+0x300];
```


Performance



Sources

XinXin Mei et. al. “Dissecting GPU Memory Hierarchy through Microbenchmarking”

Tony Scuderio, Memory Bandwidth Bootcamp: Best Practices

Tony Scuderio, Memory Bandwidth Bootcamp: Beyond Best Practices

Maxim Milakov: Fast Dynamic Indexing of Private Arrays in CUDA

Jiri Kraus, CUDA Performance Optimisation