GPU TECHNOLOGY CONFERENCE

GPU MEMORY BOOTCAMP II: BEYOND BEST PRACTICES

TONY SCUDIERO - NVIDIA

GOAL

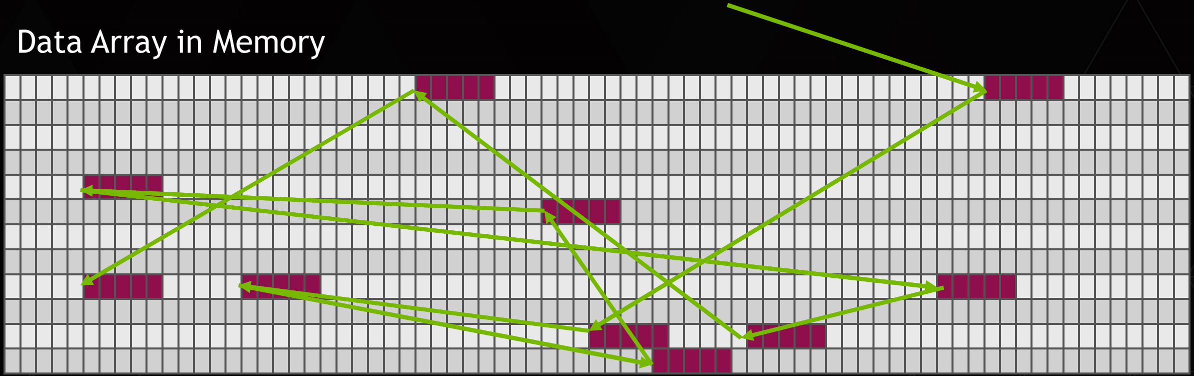
- Introduce approaches to improving memory limited codes on GPU which go beyond the standard “best practices”
- Explain why these optimizations help
- Address memory access issues in applications which are not “GPU-friendly”

DEMONSTRATION CODE

- Memory Performance Characteristics of:
  - Random Data Lookup
  - Databases
  - Analytics
  - Graph-style problems

- Motivating Example:
  - Material cross section calculation for Monte Carlo neutron transport calculations

DEMONSTRATION CODE



Access of a Single Thread

THE DATA

Parameters:

N - The number of elements in the DataArray and OffsetArray

M - The number of different “tasks.” == number of GPU Threads.

Summands - Number of sequential data elements read by the algorithm

Iterations - Parameter controlling generation of IterationArray

T - Datatype of Data Array <template parameter>

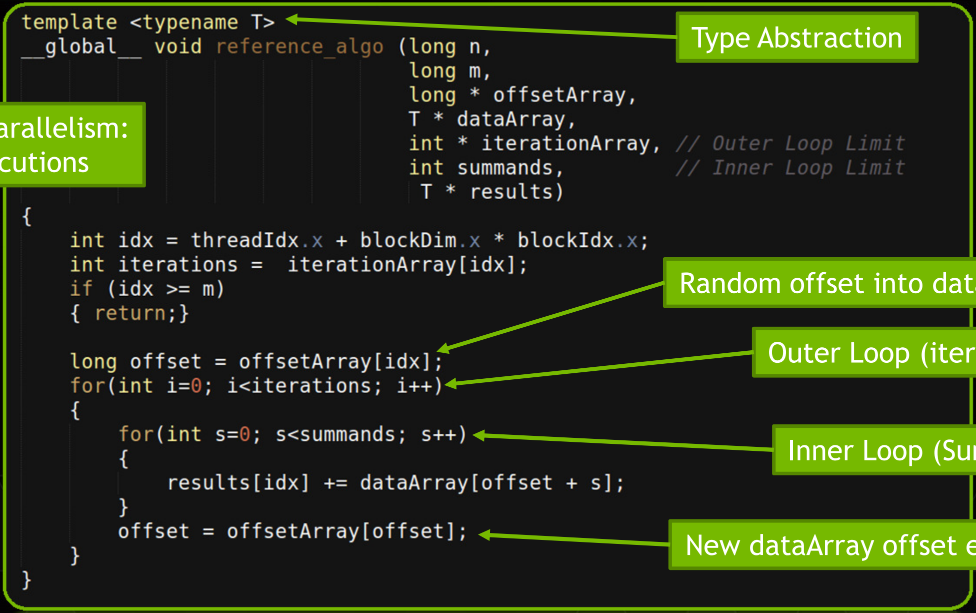
Data:

DataArray - Randomly initialized data of a datatype (C++ template)

OffsetArray - Randomly initialized array of long integers between 0 and N

IterationArray - Array of size M specifying how many iterations each thread executes

BASIC CODE



```
template <typename T>
__global__ void reference_algo (long n,
                                long m,
                                long * offsetArray,
                                T * dataArray,
                                int * iterationArray, // Outer Loop Limit
                                int summands,         // Inner Loop Limit
                                 T * results)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int iterations =  iterationArray[idx];
    if (idx >= m)
    { return;}

    long offset = offsetArray[idx];
    for(int i=0; i<iterations; i++)
    {
        for(int s=0; s<summands; s++)
        {
            results[idx] += dataArray[offset + s];
        }
        offset = offsetArray[offset];
    }
}
```

Type Abstraction

Thread Parallelism:
M executions

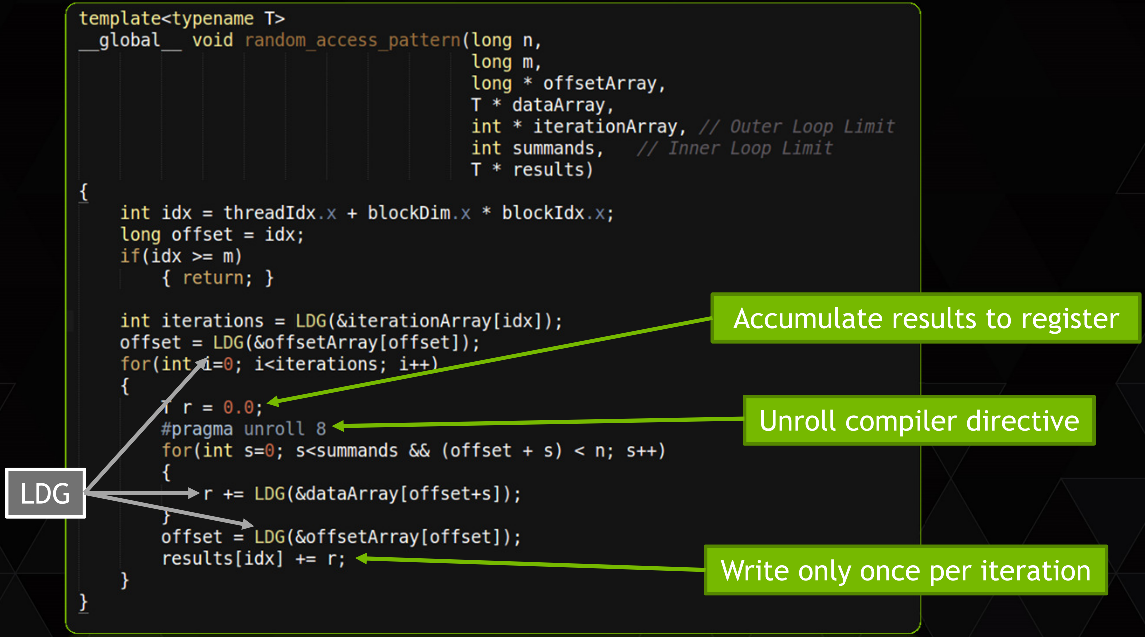
Random offset into dataArray

Outer Loop (iterations)

Inner Loop (Summands)

New dataArray offset each Iteration

ZERO ORDER OPTIMIZATION



```
template<typename T>
__global__ void random_access_pattern(long n,
                                      long m,
                                      long * offsetArray,
                                      T * dataArray,
                                      int * iterationArray, // Outer Loop Limit
                                      int summands,    // Inner Loop Limit
                                      T * results)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    long offset = idx;
    if(idx >= m)
        { return; }

    int iterations = LDG(&iterationArray[idx]);
    offset = LDG(&offsetArray[offset]);
    for(int i=0; i<iterations; i++)
    {
        T r = 0.0;
        #pragma unroll 8
        for(int s=0; s<summands && (offset + s) < n; s++)
        {
            r += LDG(&dataArray[offset+s]);
        }
        offset = LDG(&offsetArray[offset]);
        results[idx] += r;
    }
}
```

LDG

Accumulate results to register

Unroll compiler directive

Write only once per iteration

STARTING PROBLEM

- Summands: 32
- Iterations: 32
- Data Size: 10*2^20 (~10M) elements
- M: 2^20 (~1M lookups)

Kernel	float	double
Original	18.22	34.57

Units: GB/sec

K40c, boost on, ECC off
32 threads/block
M=1, N=10, S=32, I=32

OPT-IN L1 CACHING

- __ldg(&data);
    - K20, K40, K80, Maxwell
    - Load through Texture unit & try to hit in TEX cache
    - 32 Byte load granularity
    - In-SM texture L1 cache (non-coherent, read only)
    - Good if using only a segment of the cache line immediately (diverged access)
- -Xptxas="-dlcm=ca"
    - Opt-in L1 caching on K40 & K80
    - 128 Byte load granularity
    - Good if using a large portion of the cache line immediately (coalesced access)

PERFORMANCE

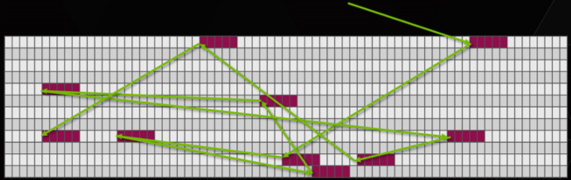
Kernel	float	double
Original	18.22	34.57
Original (L1)	22.39	44.54
Original (TEX)	44.54	56.47

Units: GB/sec

K40c, boost on, ECC off
32 threads/block
M=1, N=10, S=32, I=32

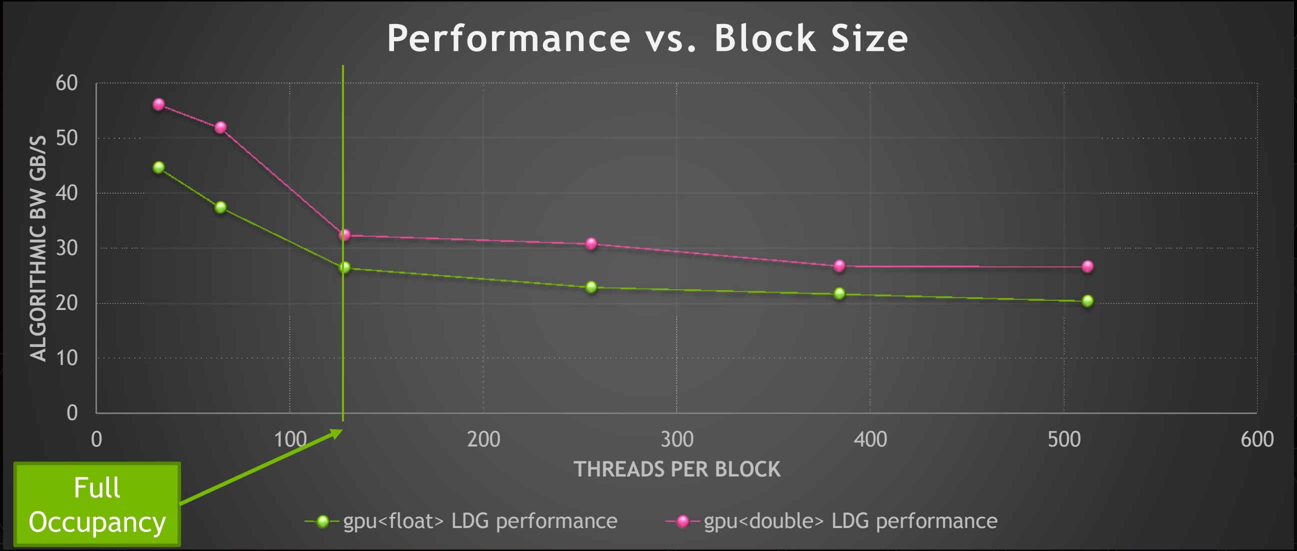
GPU TECHNOLOGY CONFERENCE

WHY CACHING SHOULD HELP:



- Single thread brings in large chunk of data
  - Subsequent loads from that thread can hit in L1 at low latency
- Why LDG/TEX performs better than L1 in this case
  - L1 brings in full cache lines(128bytes)
  - LDG brings in only necessary segments (32 bytes)
  - We are pulling the same cache lines / segments more than once, even in TEX
    - A single thread is not making it through it’s 32 or 128 bytes before they get evicted
    - Hit rate tells us this

UNFORSEEN CONSEQUENCES



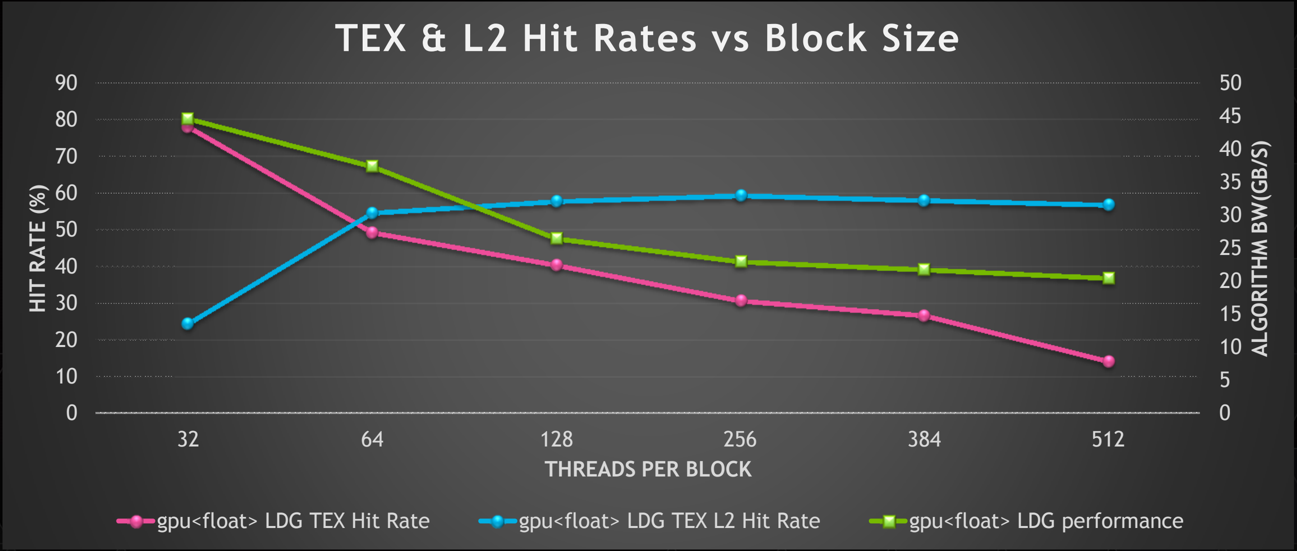
K40c, boost on, ECC off
i=32, s=32, $M=1024^2$, N=10

WHAT HAPPENED?

- Occupancy and cache per thread are contrarian measures
  - As occupancy increase, cache size remains stubbornly constant
    - => bytes of cache per thread decreases
  - Expect hit rate to fall: more competition for the same cache lines

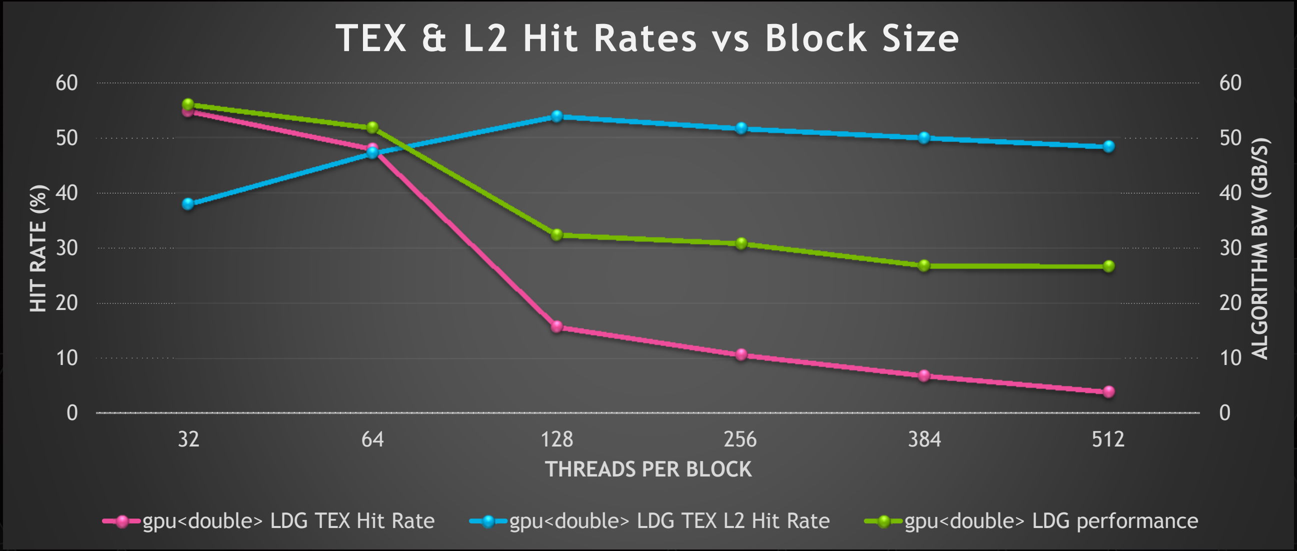
- How you can tell: NVPROF/NVVP counters
  - `l1_cache_global_hit_rate` - % of reads that hit in L1 (Opt-In, K40/K80 only)
  - `tex_cache_hit_rate` - % of reads that hit in tex cache (__ldg() or texture access only)
  - `l2_l1_read_hit_rate` - % of l1 misses which hit in L2
  - `l2_texture_read_hit_rate` - % of texture misses which hit in L2

OCCUPANCY AND CACHING <FLOAT>



K40c, boost on, ECC off
i=32, s=32, $M=1024^2$, N=10

OCCUPANCY AND CACHING <DOUBLE>



K40c, boost on, ECC off

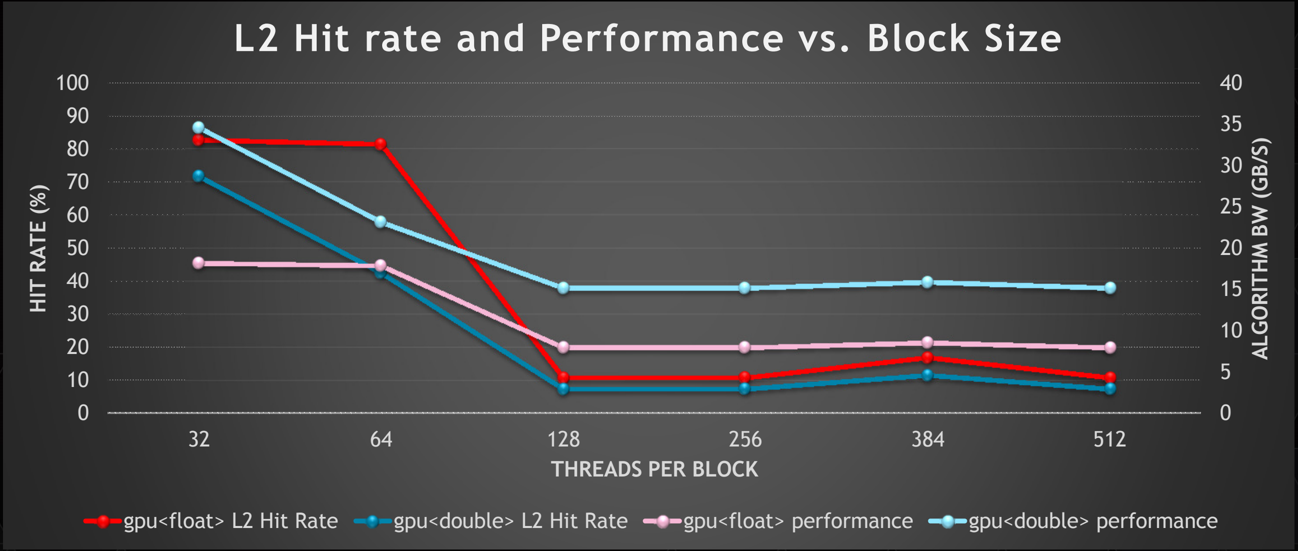
i=32, s=32, M=1024^2, N=10

OCCUPANCY AND CACHING

- According to GPU101: Latency hiding through parallelism
  - Higher Occupancy improves latency hiding
  - better latency hiding is better performance
  - ergo “Higher Occupancy has higher performance”
    - This is a broad generalization with many exceptions

- GTC2010: “Better Performance at Lower Occupancy” V. Volkov
  - With Fermi and caching, lower occupancy can be better
  - Latency hiding can be done in-thread, not just across threads

WHAT ABOUT L2 CACHING?



K40c, boost on, ECC off
i=32, s=32, M=1024^2, N=10

TL;DR; LAST 9 SLIDES

- When latency bound caching may help, try small blocksizes to increase effective cache per thread
- Use dynamic shared to limit occupancy

- Latency reduction of caching can improve performance more than latency hiding of parallelism
  - Especially if you don’t have enough compute or memory parallelism to fully hide latency
    - I.E. Traditionally non-GPU-friendly workloads

PERFORMANCE

Kernel	float	double
Original	18.22	34.57
Original (L1)	22.39	44.54
Original (TEX)	44.54	56.47

Units: GB/sec

K40c, boost on, ECC off
32 threads/block
M=1, N=10, S=32, I=32

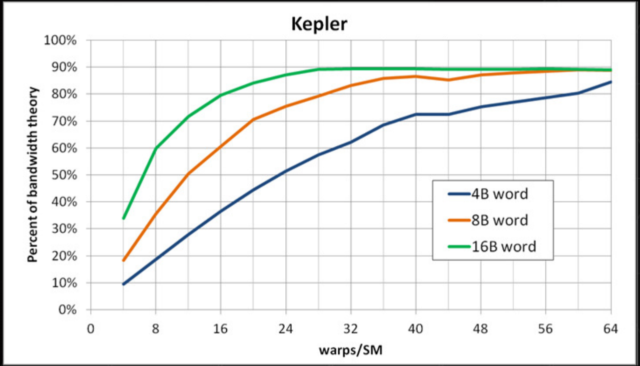
DATA LOAD SIZE

- GPU Loads: 1, 2, 4, 8, 16 bytes / thread
  - Thread View: A coalesced load of 4 bytes per thread `(float)`
    - SM View: 128 Bytes, 1 Cache line in flight/load/warp
    - L2/DRAM View: 4 memory transactions in flight/load/warp
  - Thread View: A coalesced load of 8 bytes per thread `(float2, double)`
    - SM View: 256 Bytes, 2 cache lines in flight/load/warp
    - L2/DRAM View: 8 memory transactions in flight/load/warp
  - Thread View: A coalesced load of 16 bytes per thread `(float 4, double2)`
    - SM View: 512 bytes, 4 cache lines in flight/load/warp
    - L2/DRAM View: 16 memory transactions in flight/load/warp

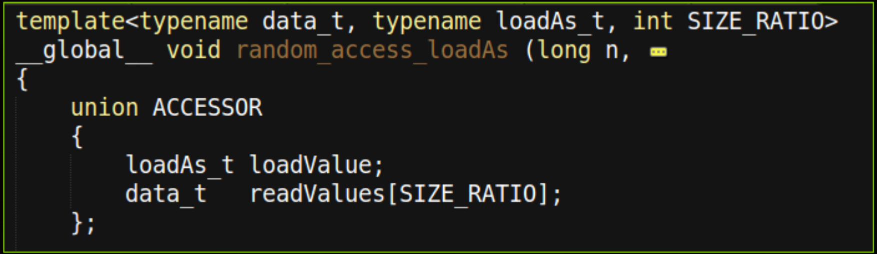
DATA LOAD SIZE

- More transactions on the bus per load
  - Fewer loads per warp are needed to saturate DRAM bandwidth
- Immediately uses larger area of the transferred line
  - More efficient use of achieved bandwidth

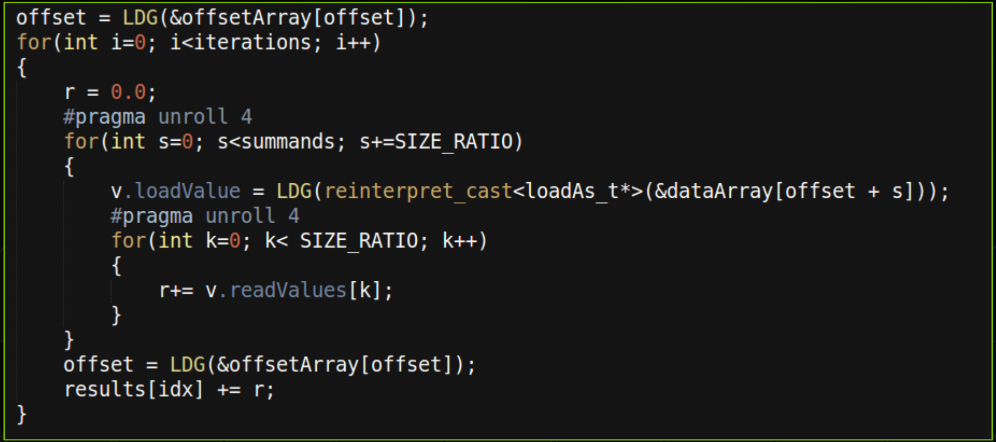
memcpy experiment
2 requests / thread experiment
Tesla K20X



LOAD AS KERNEL

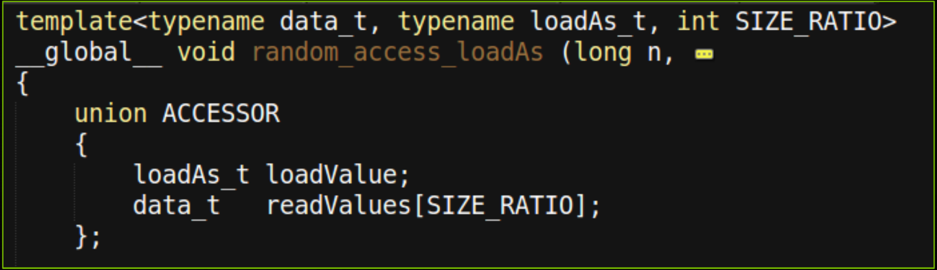


```
template<typename data_t, typename loadAs_t, int SIZE_RATIO>
__global__ void random_access_loadAs (long n, …
{
    union ACCESSOR
    {
        loadAs_t loadValue;
        data_t   readValues[SIZE_RATIO];
    };
```



```
offset = LDG(&offsetArray[offset]);
for(int i=0; i<iterations; i++)
{
    r = 0.0;
    #pragma unroll 4
    for(int s=0; s<summands; s+=SIZE_RATIO)
    {
        v.loadValue = LDG(reinterpret_cast<loadAs_t*>(&dataArray[offset + s]));
        #pragma unroll 4
        for(int k=0; k< SIZE_RATIO; k++)
        {
            r+= v.readValues[k];
        }
    }
    offset = LDG(&offsetArray[offset]);
    results[idx] += r;
}
```

USING LOAD AS



```
template<typename data_t, typename loadAs_t, int SIZE_RATIO>
__global__ void random_access_loadAs (long n, ⋯
{
    union ACCESSOR
    {
        loadAs_t loadValue;
        data_t   readValues[SIZE_RATIO];
    };
```

```
random_access_loadAs<float, float4, 4> <<<grid,thread>>>(...)
random_access_loadAs<double, double2, 2) <<<grid,thread>>>(...)
```

Caveat: Data reads must have alignment of the larger `loadAs_t` type

PERFORMANCE

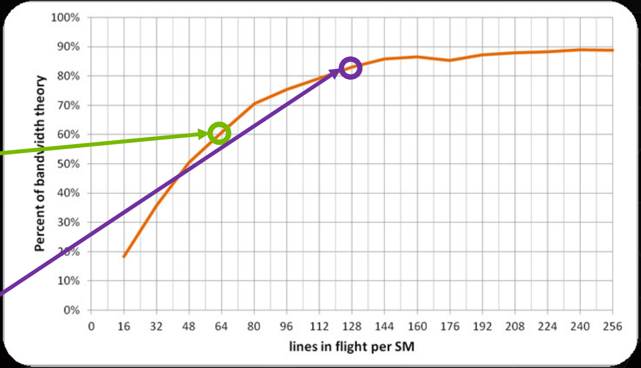
Kernel	float	float (TEX)	double	double (TEX)
Original	18.22	44.54	34.57	56.47
loadAs	42.35	45.21	50.34	52.42

Units: GB/sec

K40c, boost on, ECC off
32 threads/block
M=1, N=10, S=32, I=32

LINES IN FLIGHT

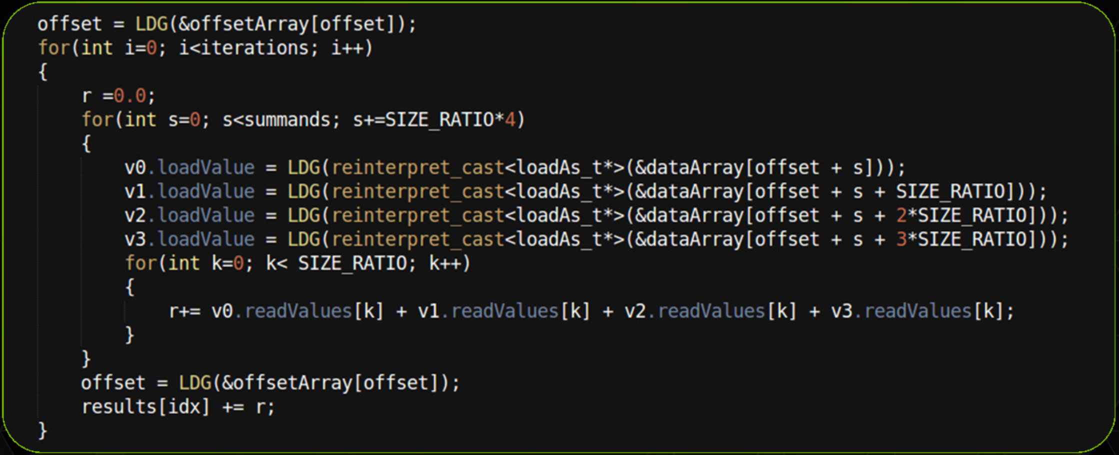
- 128-byte cache lines
    - (4 DRAM transactions/line)
- @100% occupancy,
    - 2048 threads on the SM = 64 warps
    - 1 32-bit coalesced ops/ thread (warp)
    - = 64 lines/SM
- 2 coalesced ops/thread
    - = 128 lines/SM



GPU TECHNOLOGY CONFERENCE

MEMORY LEVEL PARALLELISM

- Unroll the summand loop, issue all loads before first use (loadAs_s4)



```
offset = LDG(&offsetArray[offset]);
for(int i=0; i<iterations; i++)
{
    r =0.0;
    for(int s=0; s<summands; s+=SIZE_RATIO*4)
    {
        v0.loadValue = LDG(reinterpret_cast<loadAs_t*>(&dataArray[offset + s]));
        v1.loadValue = LDG(reinterpret_cast<loadAs_t*>(&dataArray[offset + s + SIZE_RATIO]));
        v2.loadValue = LDG(reinterpret_cast<loadAs_t*>(&dataArray[offset + s + 2*SIZE_RATIO]));
        v3.loadValue = LDG(reinterpret_cast<loadAs_t*>(&dataArray[offset + s + 3*SIZE_RATIO]));
        for(int k=0; k< SIZE_RATIO; k++)
        {
            r+= v0.readValues[k] + v1.readValues[k] + v2.readValues[k] + v3.readValues[k];
        }
    }
    offset = LDG(&offsetArray[offset]);
    results[idx] += r;
}
```

PERFORMANCE

Kernel	float	float (TEX)	double	double (TEX)
Original	18.22	44.54	34.57	56.47
loadAs	42.35	45.21	50.34	52.42
loadAs(unroll 2)	42.61	50.34	50.91	58.99
loadAs(unroll 4)	43.07	48.02	47.77	55.43

- Extra lines/transactions in flight helps, but too many causes cache evictions in TEX (hit rate decreases)

Units: GB/sec

K40c, boost on, ECC off
32 threads/block
M=1, N=10, S=32, I=32

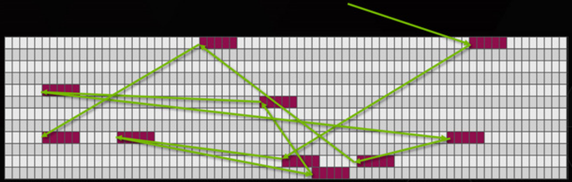
THE STORY SO FAR

- Things that helped performance:
  - Issuing through texture unit (__ldg() and running at lower occupancy
  - Issuing loads for larger datatypes
  - Issuing more loads before first use

- All of this is targeted at getting better use of the global memory bus
  - Get higher bandwidth, even when threads are limited by latency
  - Make more efficient use of the bytes we move

COMPUTE DIVERGENCE

NEW PROBLEM

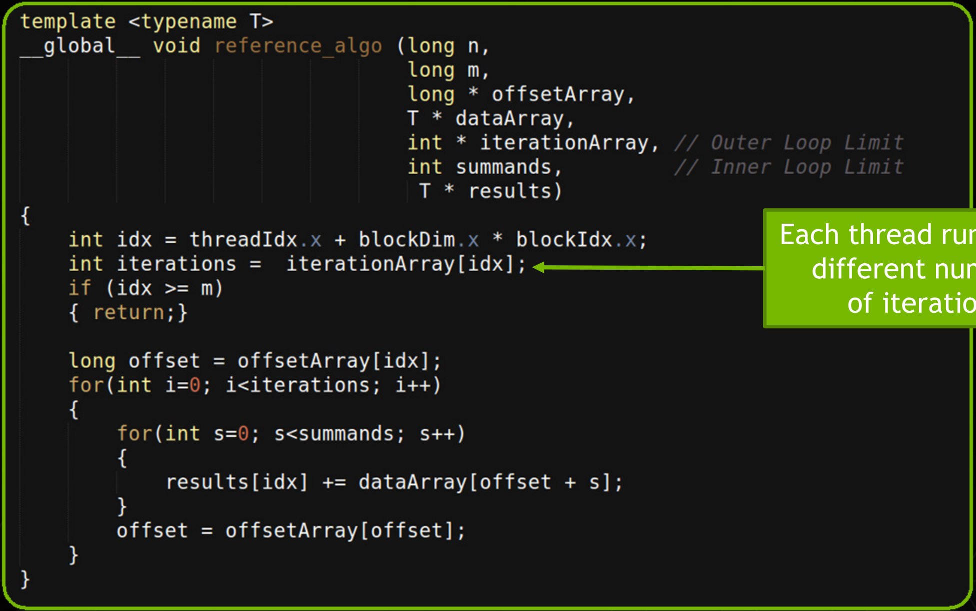


- 1M threads (M=1)
- 10M data elements (N=10)
- 32 summands per iteration (S=32)

- # of iterations uniformly distributed in [1, 128] (I=128, D=128)

- Note: Previous results are incomparable (different workloads)

IN THE CODE



```
template <typename T>
__global__ void reference_algo (long n,
                                long m,
                                long * offsetArray,
                                T * dataArray,
                                int * iterationArray, // Outer Loop Limit
                                int summands,         // Inner Loop Limit
                                 T * results)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int iterations =  iterationArray[idx];
    if (idx >= m)
    { return;}

    long offset = offsetArray[idx];
    for(int i=0; i<iterations; i++)
    {
        for(int s=0; s<summands; s++)
        {
            results[idx] += dataArray[offset + s];
        }
        offset = offsetArray[offset];
    }
}
```

Each thread runs for a different number of iterations

COMPUTE DIVERGENCE

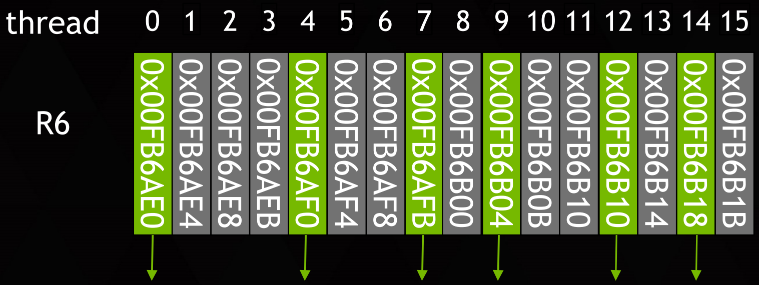
- Usually we think of divergence from FP utilization
- Think instead about memory system utilization
- Remember the warp:

```
LD.E  R2, [R6]
```

thread	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R6	0x00FB6AE0	0x00FB6AE4	0x00FB6AE8	0x00FB6AEB	0x00FB6AF0	0x00FB6AF4	0x00FB6AF8	0x00FB6AFB	0x00FB6B00	0x00FB6B04	0x00FB6B08	0x00FB6B0B	0x00FB6B10	0x00FB6B14	0x00FB6B18	0x00FB6B1B

Half of warp
shown for clarity

MEMORY OPS FROM DIVERGED WARP



thread	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R6	0x00FB6AE0	0x00FB6AE4	0x00FB6AE8	0x00FB6AEB	0x00FB6AF0	0x00FB6AF4	0x00FB6AF8	0x00FB6AFB	0x00FB6B00	0x00FB6B04	0x00FB6B0B	0x00FB6B10	0x00FB6B10	0x00FB6B14	0x00FB6B18	0x00FB6B1B

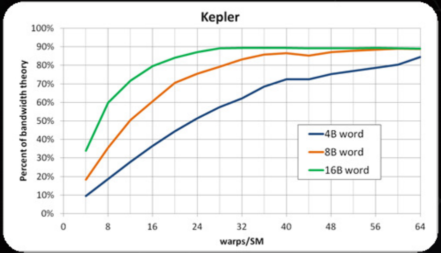
Half of warp
shown for clarity

Few addresses generated by warp
Fewer Loads in Flight
Less DRAM Bandwidth Utilization

LOADS IN FLIGHT PER SM

- When diverged, lines in flight/warp decreases
  - average dram transactions in flight decreases

- Mitigating:
  - Issue loads for larger datatypes:
    - Better use of cache lines loaded
  - Memory Level Parallelism
    - Issue multiple independent memory operations
    - More transactions in flight per thread
  - Things we've already done in the unrolled loadAs kernel



DIVERGED COMPUTATION PERFORMANCE

Kernel	float	float (TEX)	double	double (TEX)
Original	19.61	45.55	35.70	51.65
loadAs	43.23	46.72	51.64	55.90
loadAs(unroll 2)	45.17	59.96	52.42	64.84
loadAs(unroll 4)	44.93	53.27	52.30	54.55

Units: GB/sec

K40c, boost on, ECC off
32 threads/block
M=1, N=10, S=32, I=128 D=128

IN SUMMARY

- Access patterns to memory
  - Memory Divergence
  - Compute Divergence

- Optimizing access
  - Use of texture unit for on-SM caching and fine grain access
  - Interaction of caching and occupancy
  - Use of large data types to increase efficiency / loads in flight
  - Improves performance of divergent codes

GPU TECHNOLOGY CONFERENCE

QED

JOIN THE CONVERSATION

#GTC15