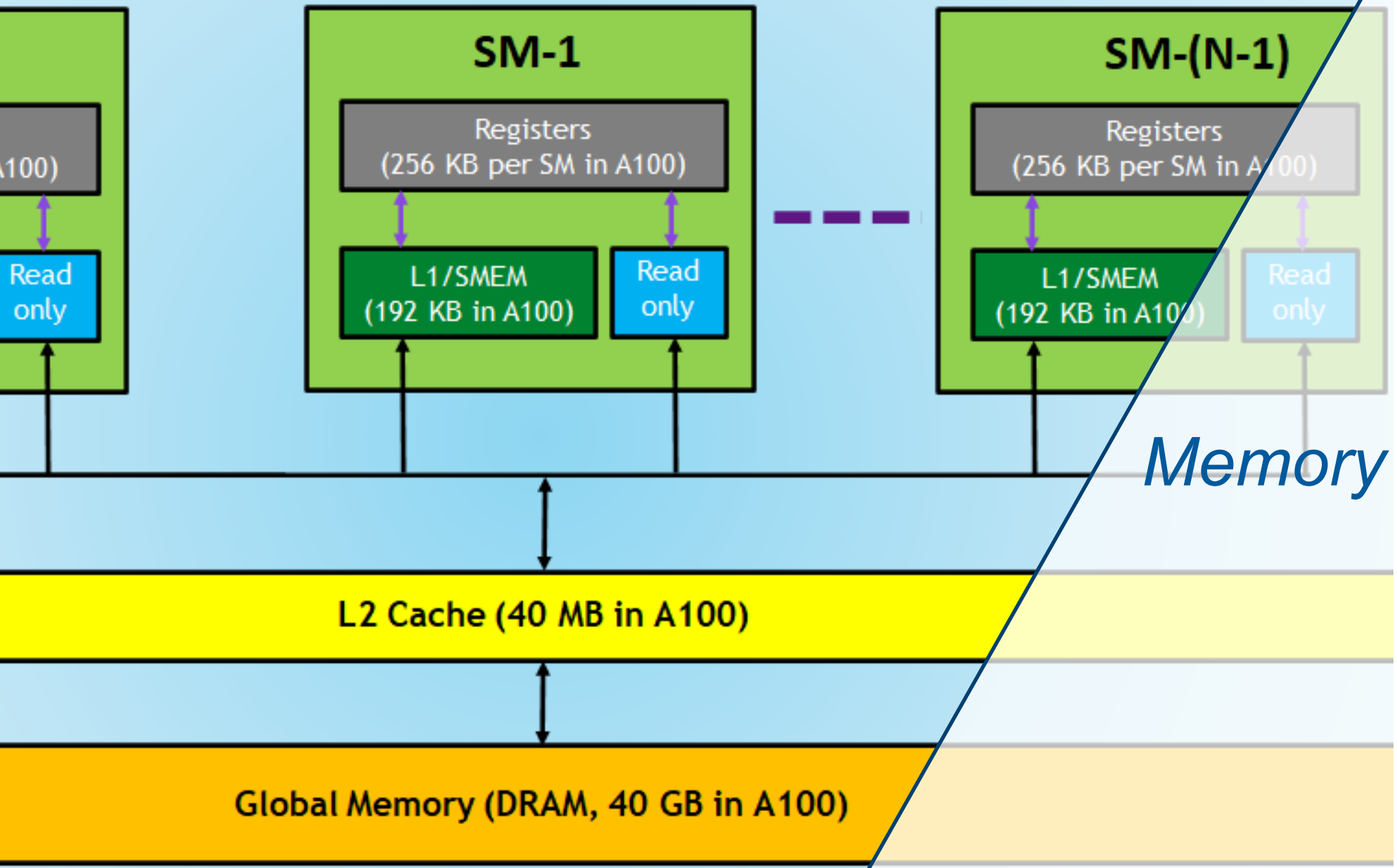
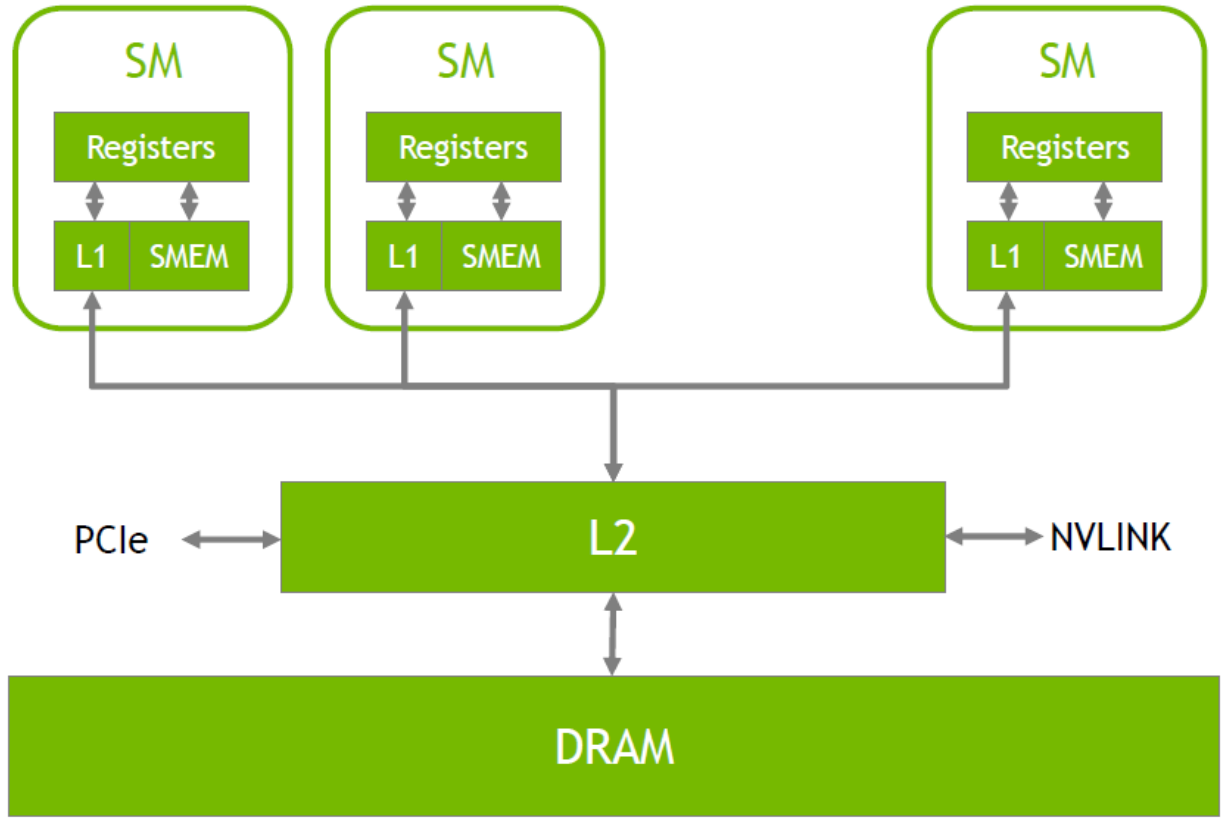
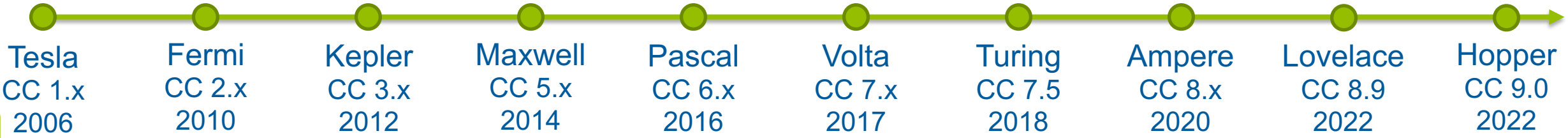


Memory Hierarchy

CUDA GPU with Caches



GPU Architecture Timeline

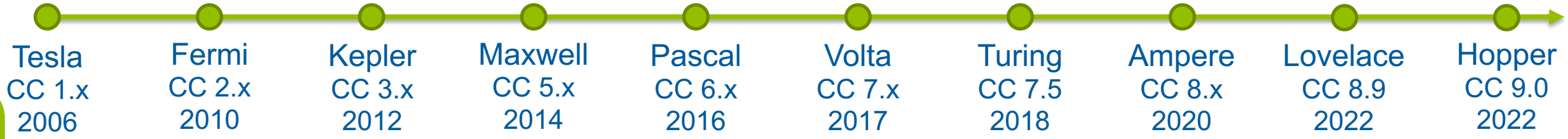


- We use a Quadro RTX 4000
  - Turing architecture TU104
  - CC 7.5
    - Consumer lineup: TU102, TU104, TU106, TU116, TU117

$CC = 7.5$

- Volta architecture
  - CC 7.0 TITAN V, Quadro GV100
    - Workstation / datacenter hardware: GV100
  - CC 7.5: Tegra Xavier, Jetson Xavier NX, DRIVE AGX Xavier, …
    - Mobile hardware: GV10B, GV11B

GPU Architecture Timeline



- We also have a GeForce GTX 970
  - Maxwell architecture GM204
  - CC 5.2

$CC = 5.2$

- Other Maxwell compute capabilities
  - CC 5.0: GeForce GTX 960M, …
    - First generation hardware: GM107, GM108
  - CC 5.2: GeForce GTX Titan X, …
    - Second generation hardware: GM200, GM204, GM206
  - CC 5.3: Tegra X1, Jetson TX1, …
    - Mobile hardware: GM20B

GPU Architecture Timeline

Tesla	Fermi	Kepler	Maxwell	Pascal	Volta	Turing	Ampere	Hopper
CC 1.x	CC 2.x	CC 3.x	CC 5.x	CC 6.x	CC 7.x	CC 7.5	CC 8.x	CC 9.0
2006	2010	2012	2014	2016	2017	2018	2020	Lovelace
								2022

Architecture	GPU Model	Compute Capability	Important Features
Tesla	GeForce GTX 8800	1.0 – 1.3	unified shaders
Fermi	GeForce GTX 480	2.0 – 2.1	ballots, 32bit floating point atomics, 3D grids
Kepler	GeForce GTX 780	3.0 – 3.7	shuffle, unified memory, dynamic parallelism
Maxwell	GeForce GTX 980	5.0 – 5.3	16-bit floating point operations (half-precision)
Pascal	GeForce GTX 1080	6.0 – 6.2	64-bit floating point atomics
Volta	Titan V	7.0 – 7.2	independent thread scheduling, tensor cores
Turing	GeForce RTX 2080	7.5	more concurrency, RTX cores (not compute)
Ampere	GeForce RTX 3090	8.0 – 8.6	L2 cache residency management
Lovelace	GeForce RTX 4090	8.9	thread block clusters, distributed shared memory
Hopper	H100	9.0	

Memory Spaces and Performance

- Global memory
  - Cached (L2 and L1)
  - Read and write
  - Dynamic allocation in device code possible
- Constant memory
  - Cached
  - Read-only
- Texture memory
  - Cached
  - Interpolation
  - ‘Read-only’
- Shared memory
  - Communication between threads
- Register
  - Private for each thread
  - (exchangeable)

Global Memory

- Main data storage
  - Use for input and output data
  - Linear arrays
- Access pattern important
- Cache behavior important
- Relatively slow
  - Bandwidth: ~ 300-700 GB/s (GDDR5/6 vs. HBM2)
  - Non-cached coalesced access: 375 cycles/ns
  - L2 cached access: 190 cycles/ns
  - L1 caches access: 30 cycles/ns

Cache Usage

- Not intended for the same use as CPU caches
  - Smaller size (especially per thread)
  - Not aimed for temporal reuse
  - Smooth-out access patterns
  - Help with spilled registers (L1 on Kepler)
- Don’t try cache blocking like on CPU
  - 100s of threads accessing L1
  - 1000s of threads accessing L2
  - Use shared memory instead
- Optimizations should not target the cache architecture

Global Memory Operations

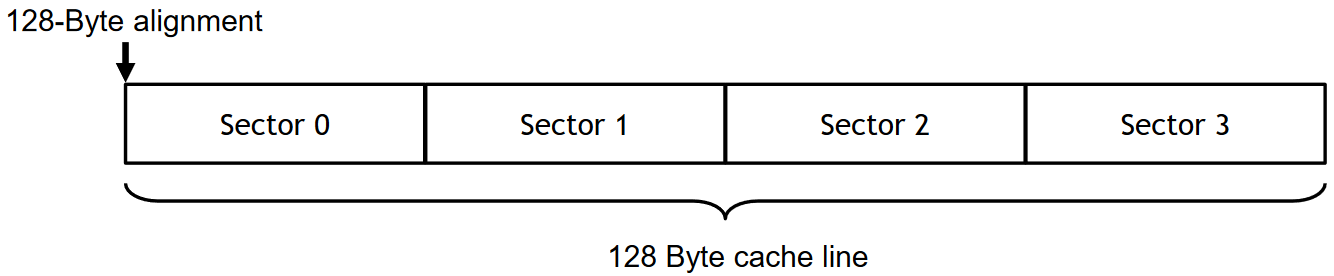
- Memory access granularity / cache line size
  - L1 & L2: 32 Byte / 128 Byte (4 sectors of 32 byte)
- Stores
  - Write-through for L1
  - Write-back for L2
- Memory operations are issued per warp
  - Threads provide addresses
  - Combined to lines/segments needed
  - Requested and served
- Try to get coalescing per warp
  - Align starting address
  - Access within a contiguous region

L1, L2 CACHES

Cache Lines & Sectors

Memory access granularity = **32 Bytes** = **1 sector**

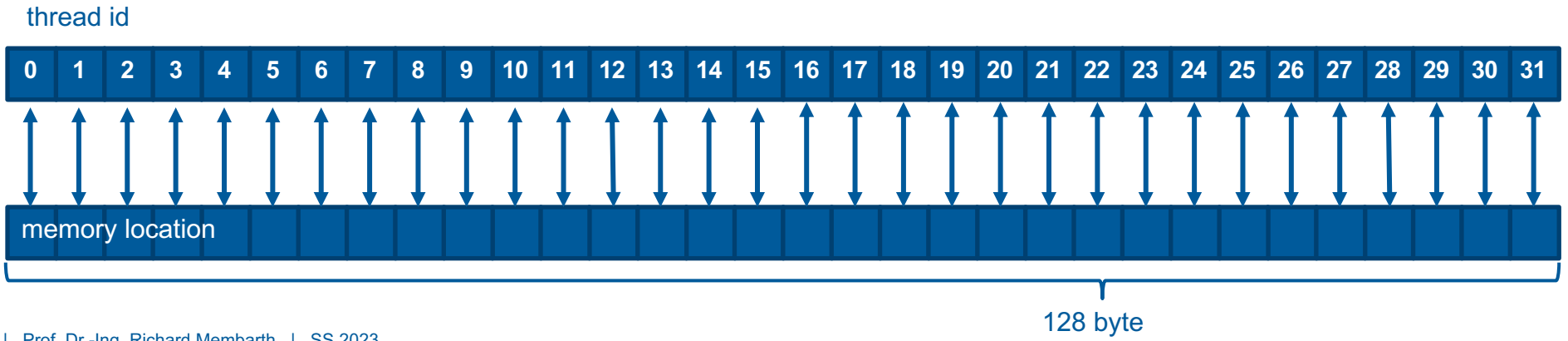
An L1/L2 **cache line** **is 128 Bytes**, made of **4 sectors**.
Cache "management" granularity = 1 cache line



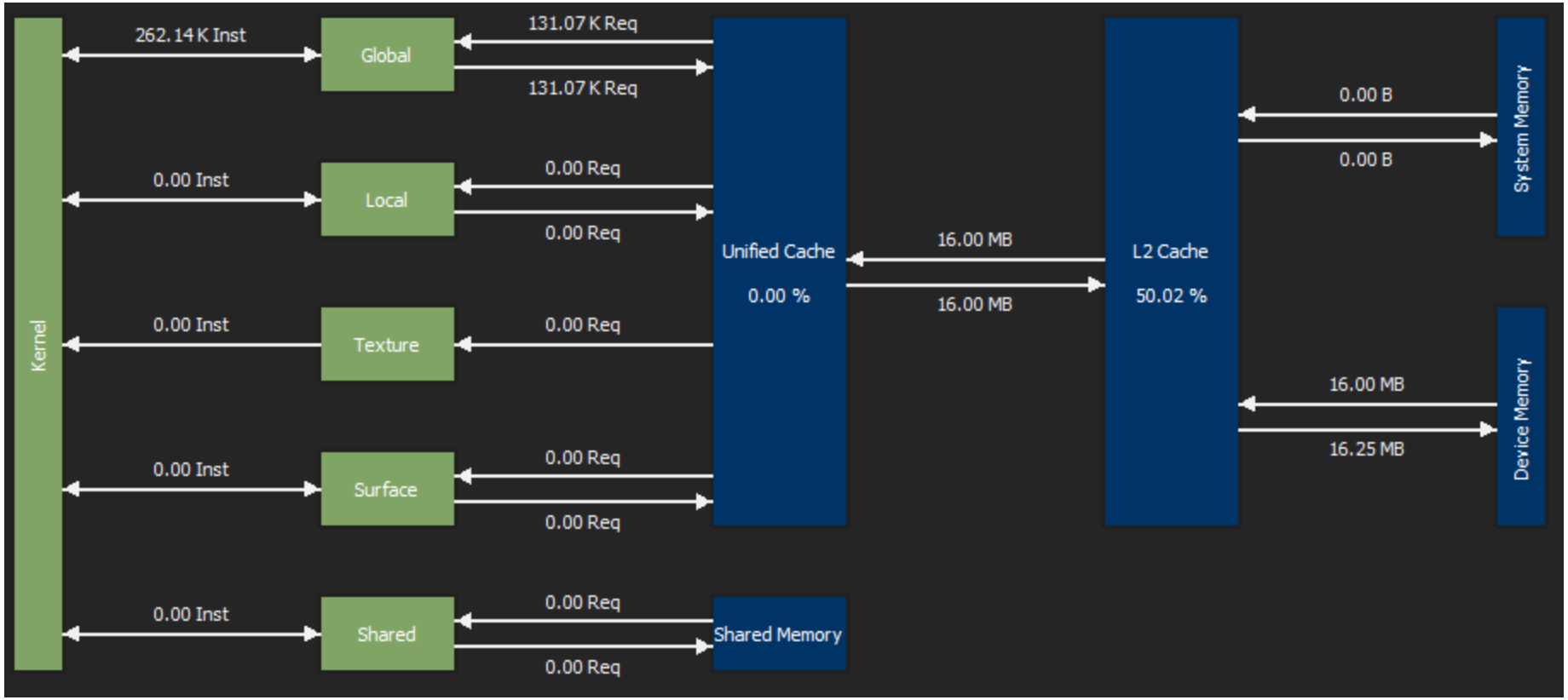
Granularity Example 1/4

```
__global__ void test_coalesced(const int* in, int* out, int elements) {
    int block_offset = blockIdx.x*blockDim.x;
    int warp_offset = 32 * (threadIdx.x / 32);
    int laneid = threadIdx.x % 32;
    int id = (block_offset + warp_offset + laneid) % elements;

    out[id] = in[id];
}
```

```
clubber:gpuprog > ./test_coalesced
test_coalesced done in 0.065568 ms -> 255.875 GB/s
```



Granularity Example 1/4

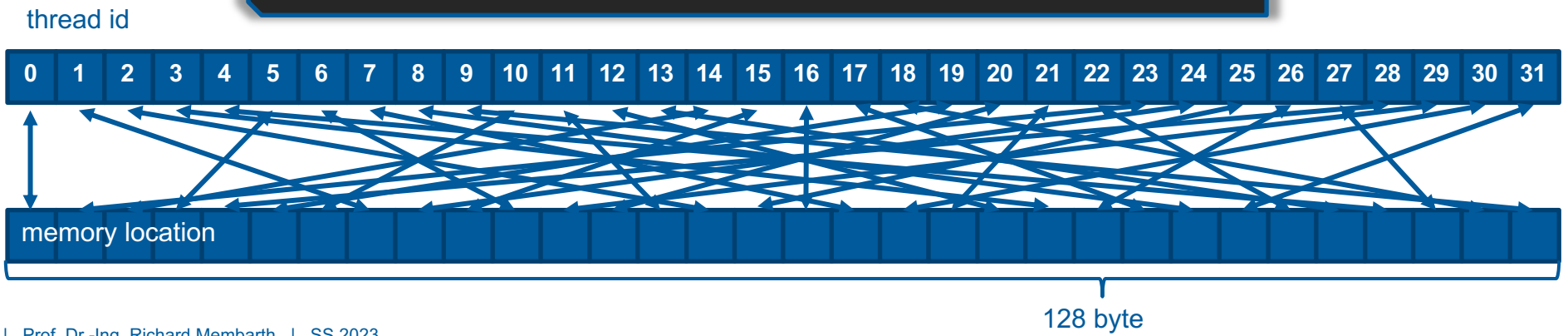


Coalesced

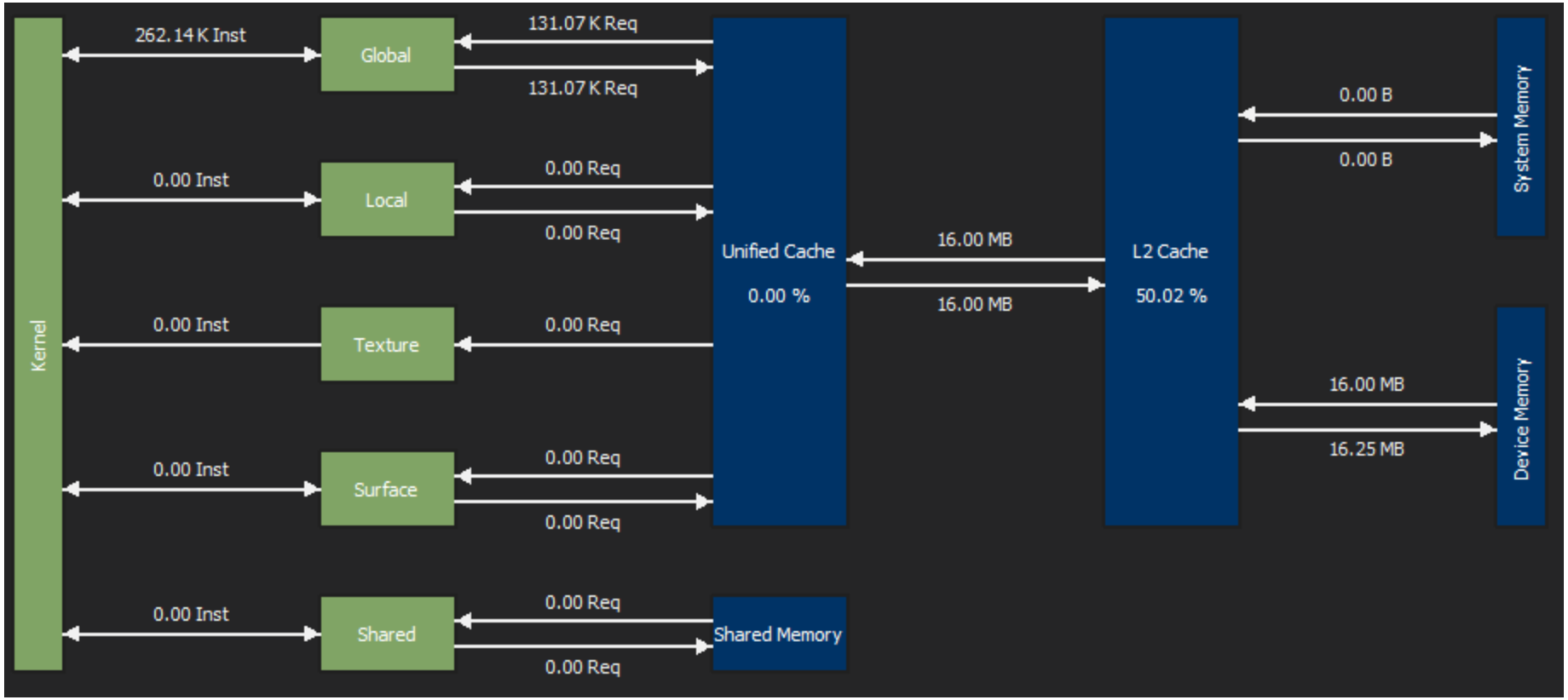
Granularity Example 2/4

```
__global__ void test_mixed(const int* in, int* out, int elements) {
    int block_offset = blockIdx.x*blockDim.x;
    int warp_offset = 32 * (threadIdx.x / 32);
    int elementid = (threadIdx.x * 7) % 32;
    int id = (block_offset + warp_offset + elementid) % elements;

    out[id] = in[id];
}
```

```
clubber:gpuprog > ./test_coalesced
test_coalesced done in 0.065568 ms -> 255.875 GB/s
test_mixed done in 0.068128 ms -> 246.26 GB/s
```



Granularity Example 2/4



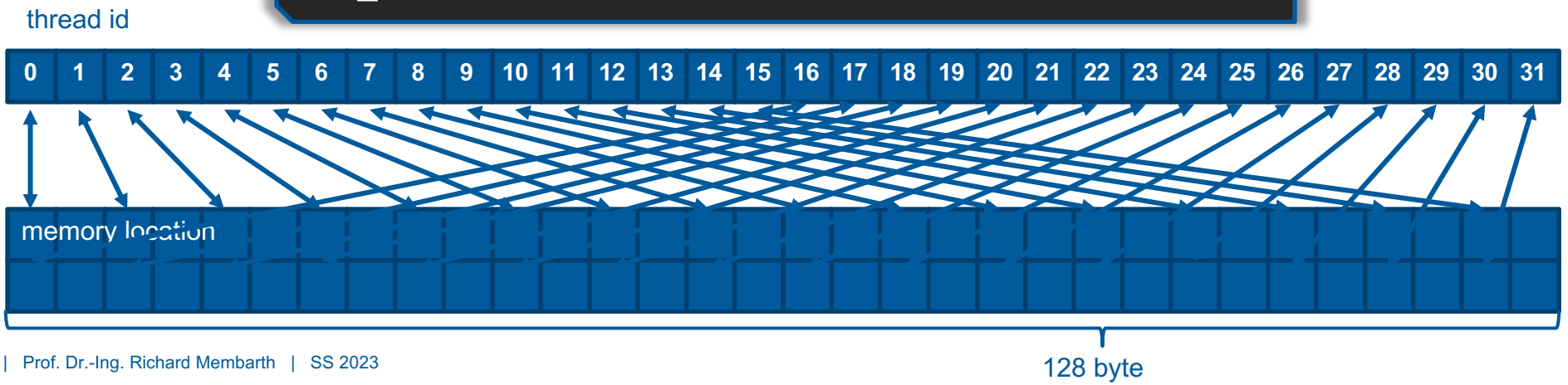
Mixed

Identical to coalesced

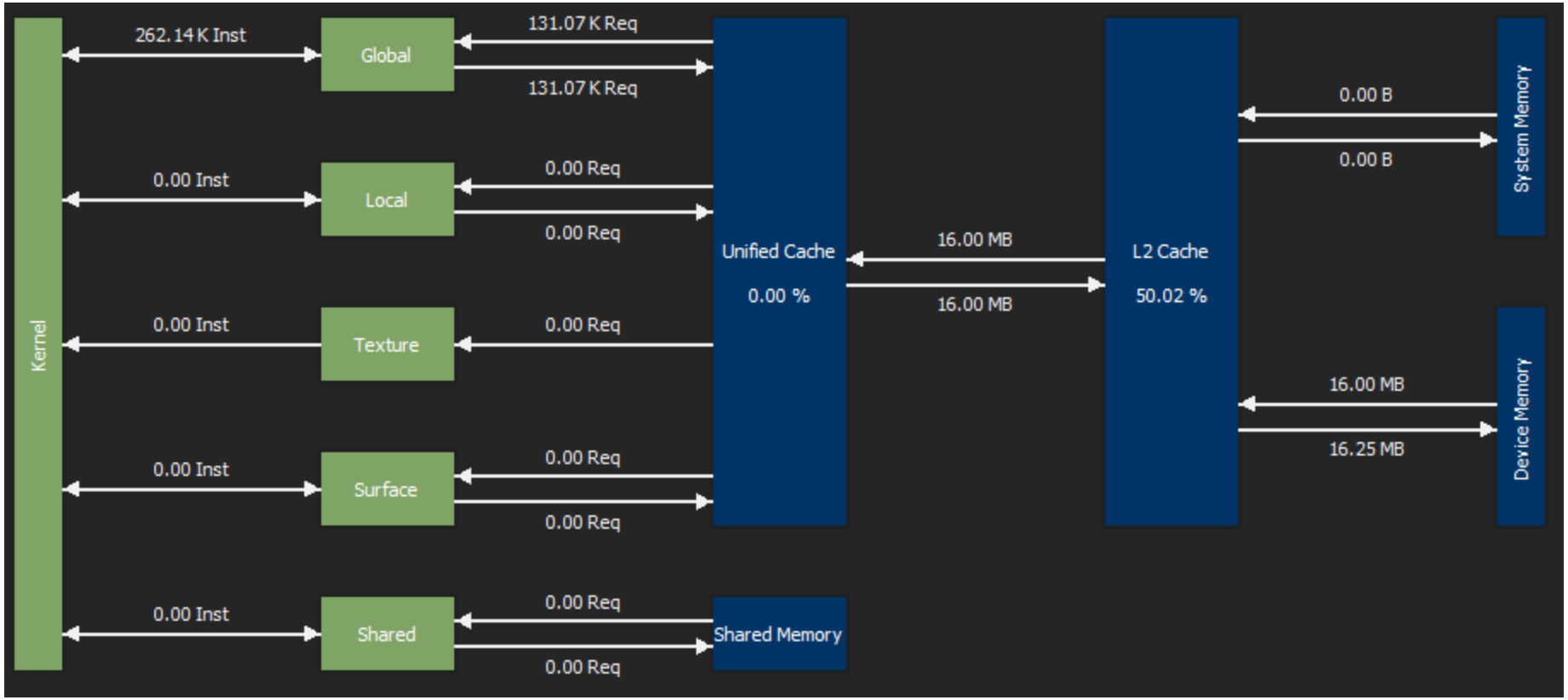
Granularity Example 3/4

```
template<int OFFSET>
__global__ void test_offset(const int* in, int* out, int elements) {
    int block_offset = blockIdx.x*blockDim.x;
    int warp_offset = 32 * (threadIdx.x / 32);
    int laneid = threadIdx.x % 32;
    int id = ((block_offset + warp_offset + laneid) * OFFSET) % elements;

    out[id] = in[id];
}
```

```
clubber:gpuprog > ./test_offset
test_coalesced -> 255.875 GB/s
test_offset<2> -> 134GB/s testOffset<4> -> 72.9GB/s
test_offset<8> -> 36.8GB/s testOffset<32> -> 17.2GB/s
```

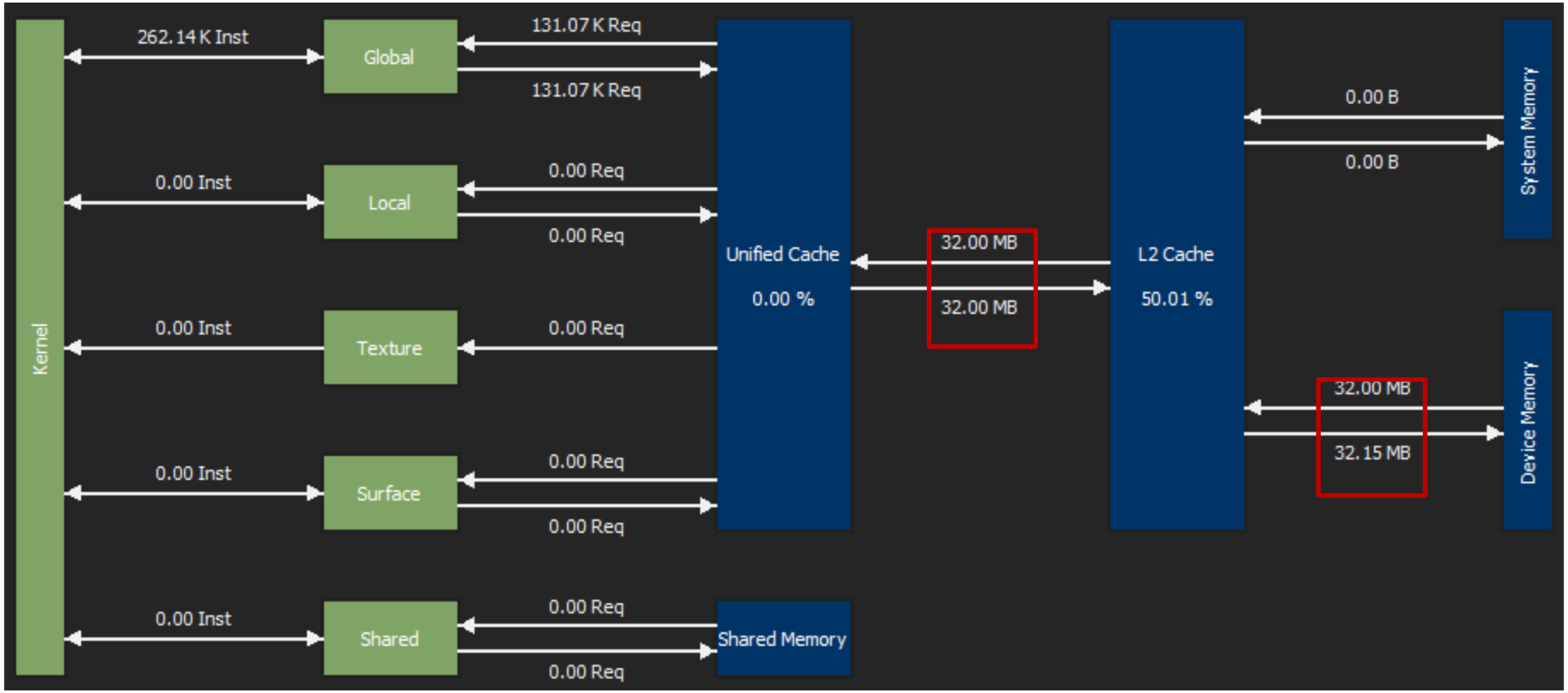


Granularity Example 3/4



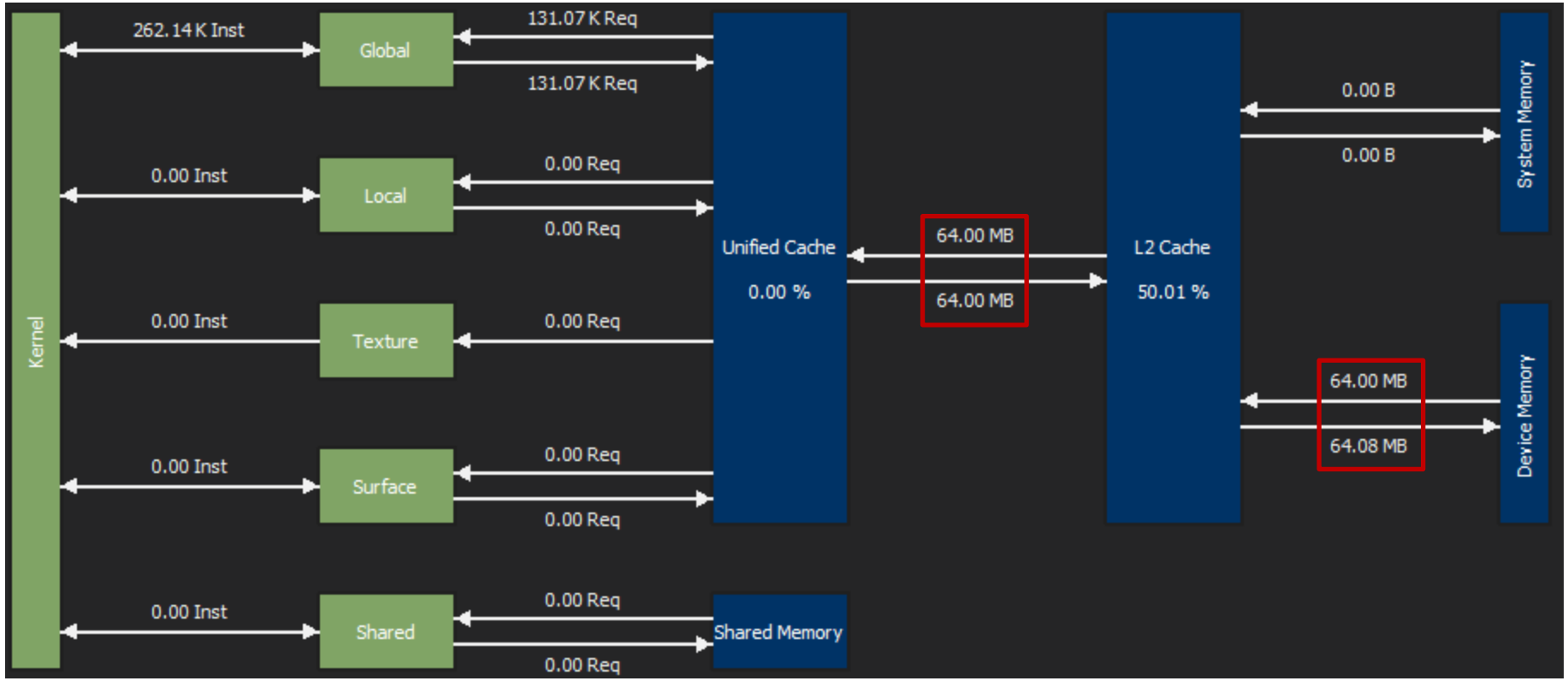
Coalesced 255.875 GB/s

Granularity Example 3/4



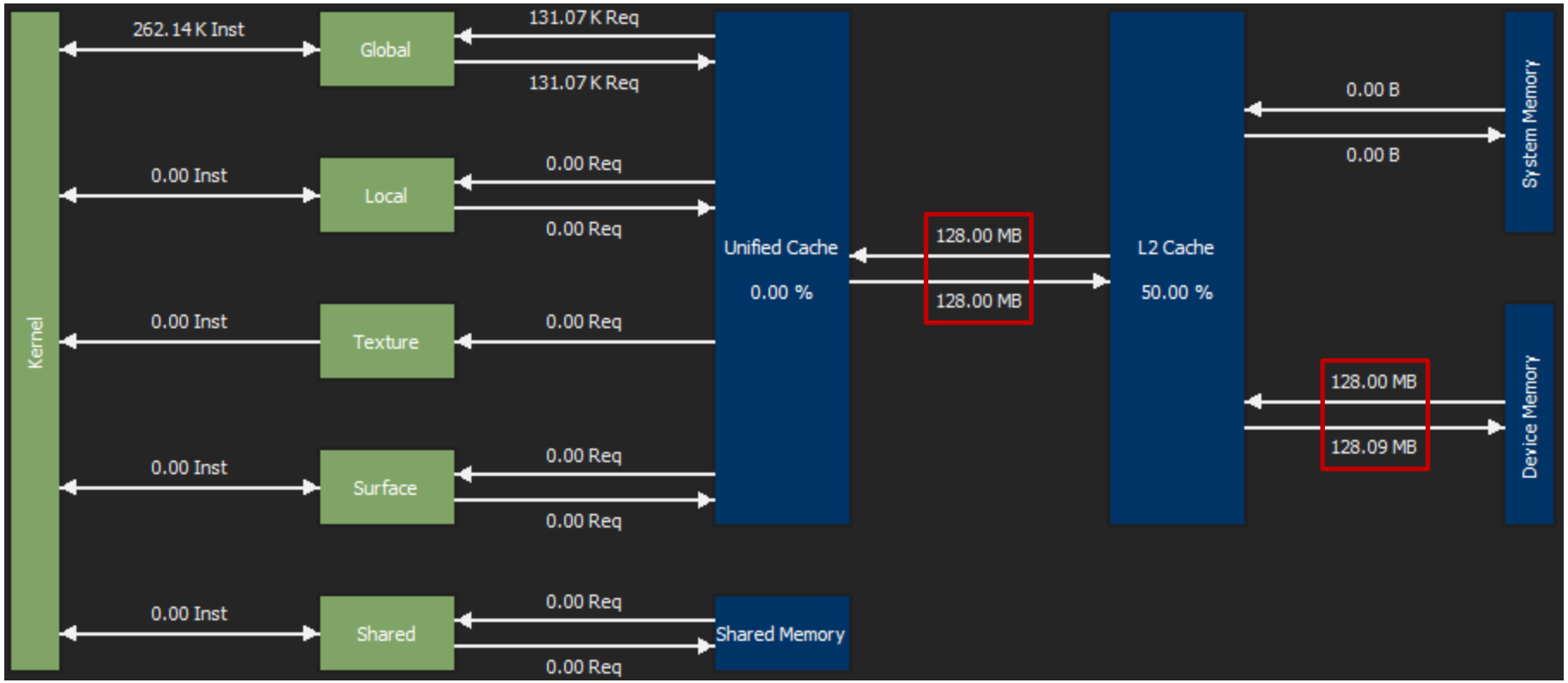
Offset 2 134.123GB/s (0.524)

Granularity Example 3/4



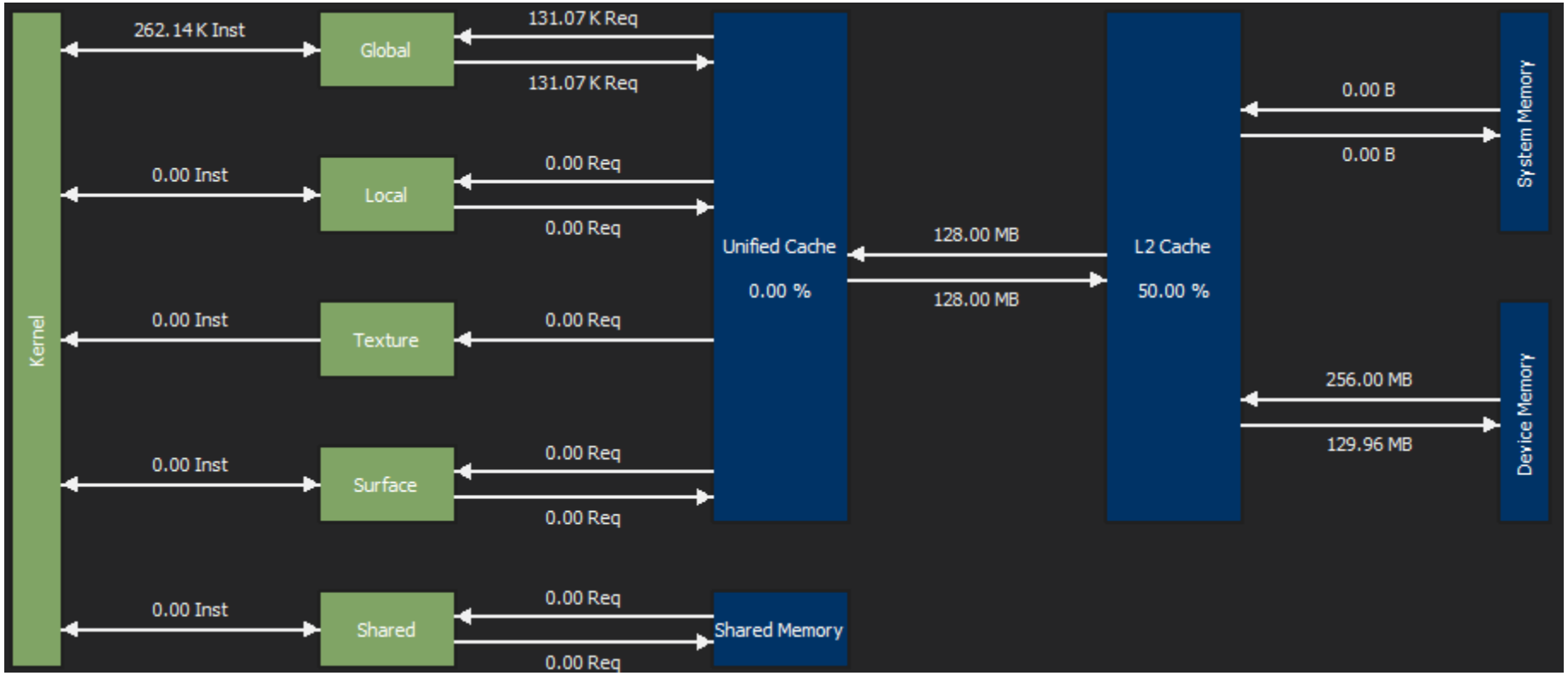
Offset 4 72.9GB/s (0.285)

Granularity Example 3/4



Offset 8 36.81GB/s (0.144)

Granularity Example 3/4



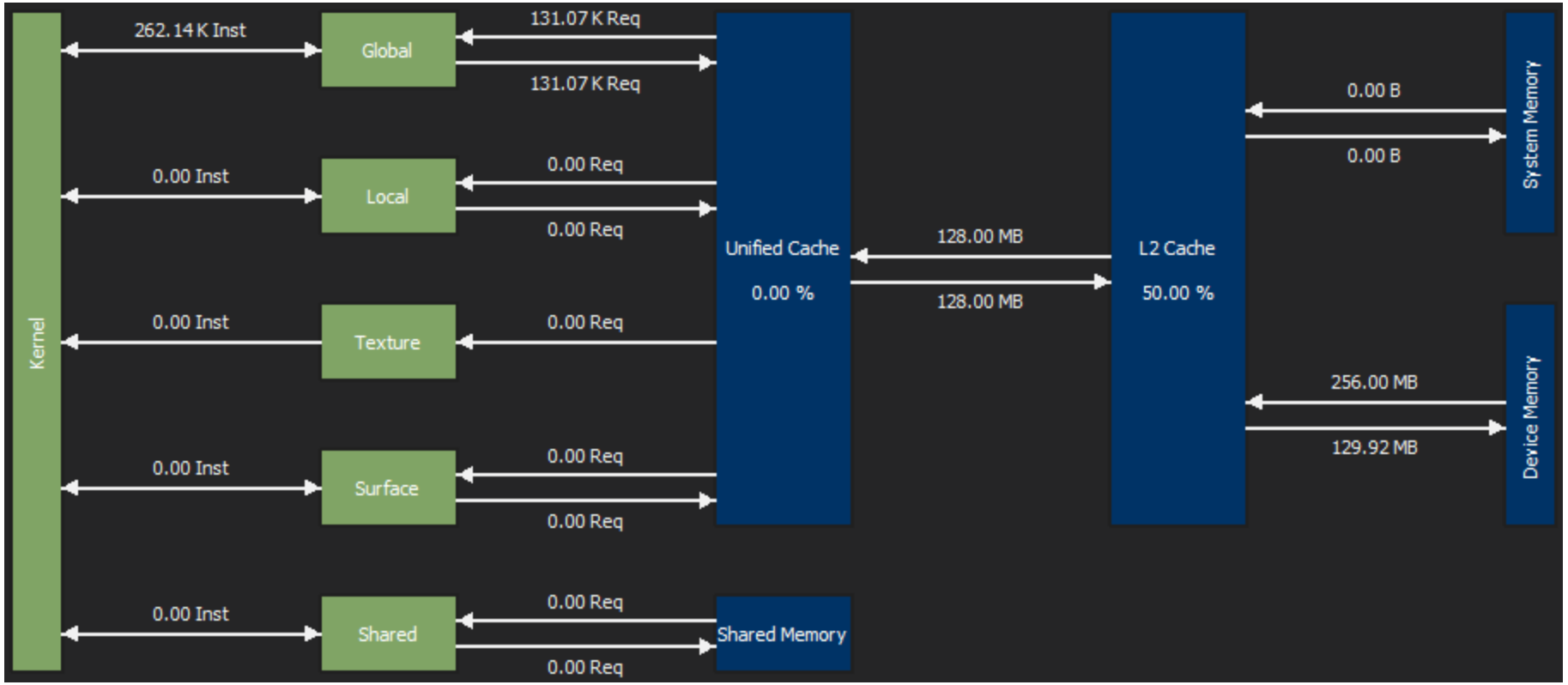
Offset 32 17.26GB/s (0.07)

Granularity Example 4/4

```
__global__ void test_scattered(const int* in, int* out, int elements) {
    int block_offset = blockIdx.x*blockDim.x;
    int warp_offset = 32 * (threadIdx.x / 32);
    int elementid = threadIdx.x % 32;
    int id = ((block_offset + warp_offset + elementid) * 121) % elements;

    out[id] = in[id];
}
```

```
clubber:gpuprog > ./test_scattered
test_coalesced done in 0.0655ms -> 255.875GB/s
test_offset<32> done in 0.972ms -> 17.2605GB/s
test_scattered done in 2.0385ms -> 8.23GB/s
```

Granularity Example 4/4



Scattered 8.23GB/s (0.03)

Granularity Example Summary

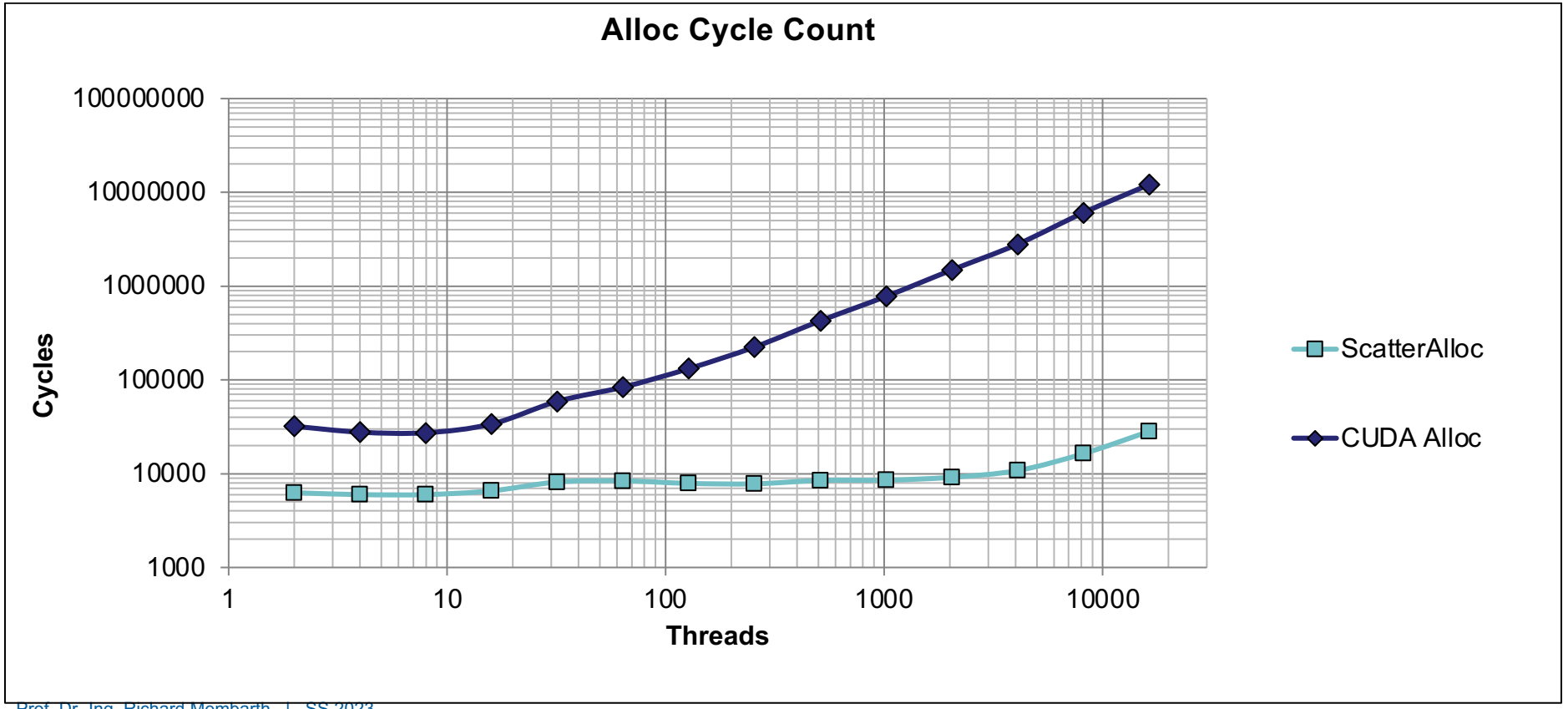
- test_coalesced done in 0.065568ms → 255.875GB/s
  test_mixed done in 0.068128ms → 246.260GB/s
  test_offset<2> done in 0.125088ms → 134.123GB/s
  test_offset<4> done in 0.229888ms → 72.979GB/s
  test_offset<8> done in 0.455744ms → 36.812GB/s
  test_offset<32> done in 0.972ms → 17.260GB/s
  test_scattered done in 2.03853ms → 8.230GB/s

- Access pattern within 128-byte segment does not matter
- Offset between data → more requests need to be handled
- Peak performance not met due to computation overhead
- More scattered data access slower with GDDR RAM

Dynamic Memory Allocation

$CC \geq 2.0$

- malloc / free, new / delete are possible
- Very slow 20 000 – 10 000 000 cycles



Constant Memory

- Read-only
  - Ideal for coefficients and other data that is read uniformly by warps
- Data is stored in global memory, read through cache
  - `__constant__` qualifier
  - Limited to 64 kB
- Supports broadcasting
  - All threads read same value → data broadcasted to all threads simultaneously
  - Otherwise diverged

Constant Memory

- Cache throughput
  - 4 bytes per warp per clock
  - All threads in warp read the same address
  - Otherwise serialized

```
__constant__ float myarray[128];

__global__ void kernel() {
    ...
    float x = myarray[23];                // uniform
    float y = myarray[blockIdx.x + 2];    // uniform
    float z = myarray[threadIdx.x];       // non-uniform
}
```

Constant Memory Example

```
__constant__ float c_array[128];

__global__ void kernel(float* d_array,
                       const __restrict__ float* dc_array) {
    float a = c_array[0];               // const cache  24cyc
    float b = c_array[blockIdx.x];      // const cache  24cyc
    float c = c_array[threadIdx.x];     // const cache  35cyc
    float d = d_array[blockIdx.x];      // L2 cache     24cyc
    float e = d_array[threadIdx.x];     // L2 cache     26cyc
    float f = dc_array[blockIdx.x];     // L1 cache     23cyc
    float g = dc_array[threadIdx.x];    // L1 cache     24cyc
}
```

Constant Memory

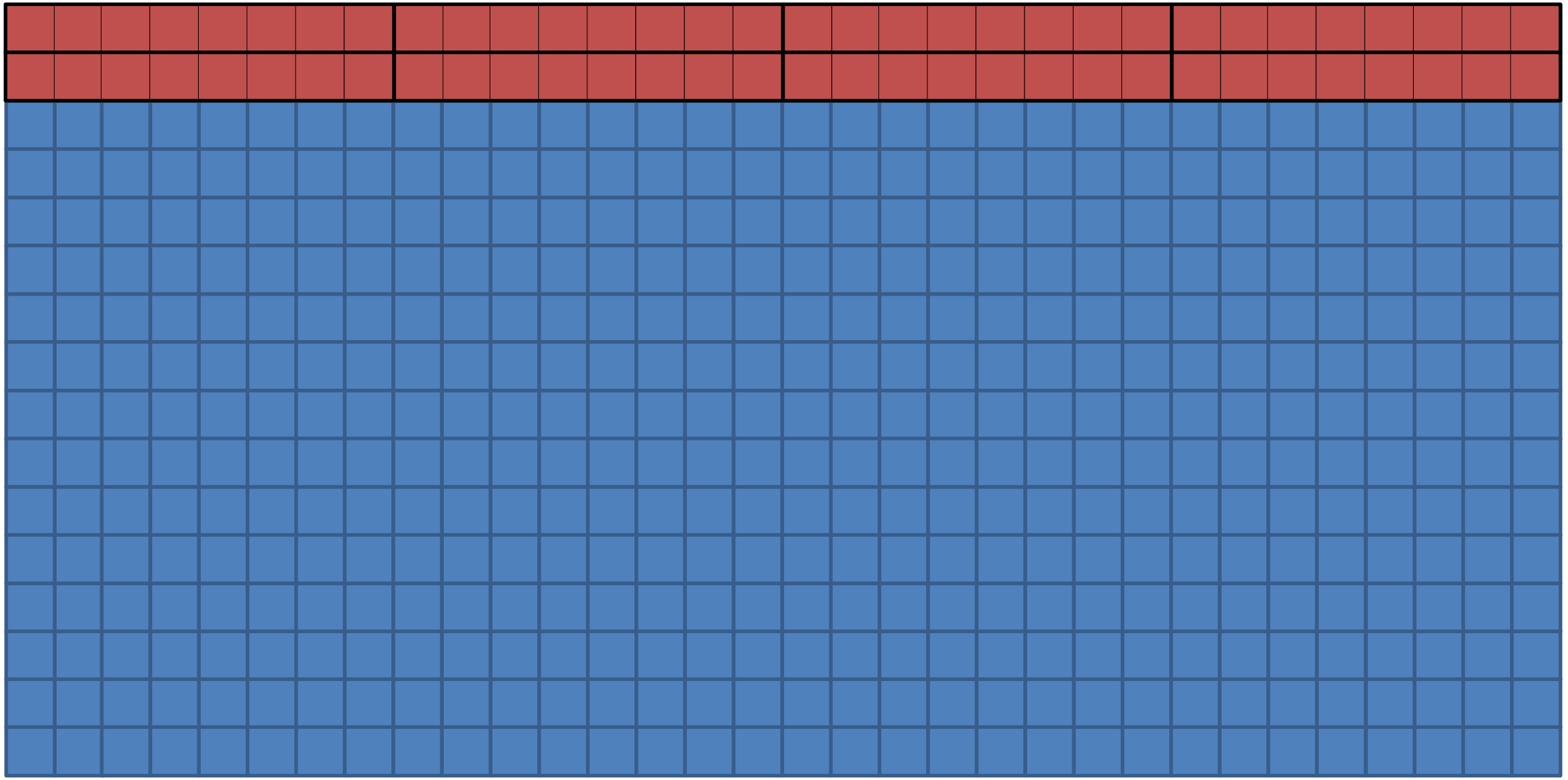
- Only fast if all threads within a warp read the same value
- Constant can be faster than global
- Uses different cache than global
- Compiler can automatically put things to constant

Texture Memory

- Textures are laid out in memory to optimize data locality for the respective dimensionality (`cudaArray`)
  - e.g., accesses to 2D region should hit the cache
- Surfaces allow writing to `cudaArrays`
- Concurrent writing and reading undefined result
  - Surfaces allow writes from kernel, but not concurrent writes/reads
- Can also bind global memory to textures

- Data that is read-only in a kernel can be cached in the unified L1 / texture cache by
  - Using both the `const` and `__restrict__` qualifier for kernel parameters
  - Using the `__ldg()` when reading from global memory

RowAfterRow – 1D



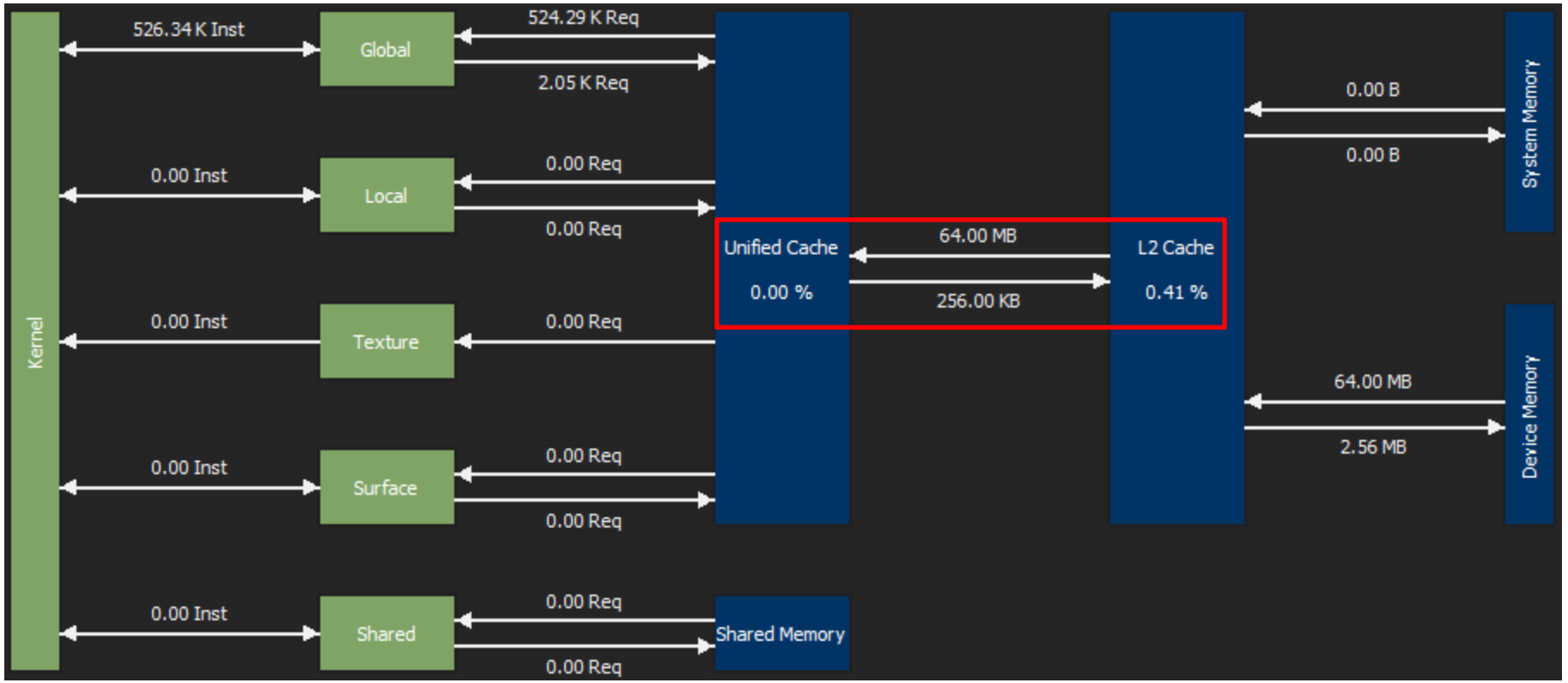
RowAfterRow: Texture vs Global

```
__global__ void device_load_row_after_row(const uchar4* data, int width, int height, uchar4* out) {
    uchar4 sum = make_uchar4(0, 0, 0, 0);

    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int rowid{0}, colid{0};

    while (tid < (height * width)) {
        rowid = tid / width;
        colid = tid % width;
        uchar4 in = data[rowid * width + colid];
        sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
        tid += blockDim.x*gridDim.x;
    }
    out[blockIdx.x*blockDim.x + threadIdx.x] = sum;
}
```

```
const uchar4* __restrict__ data
```

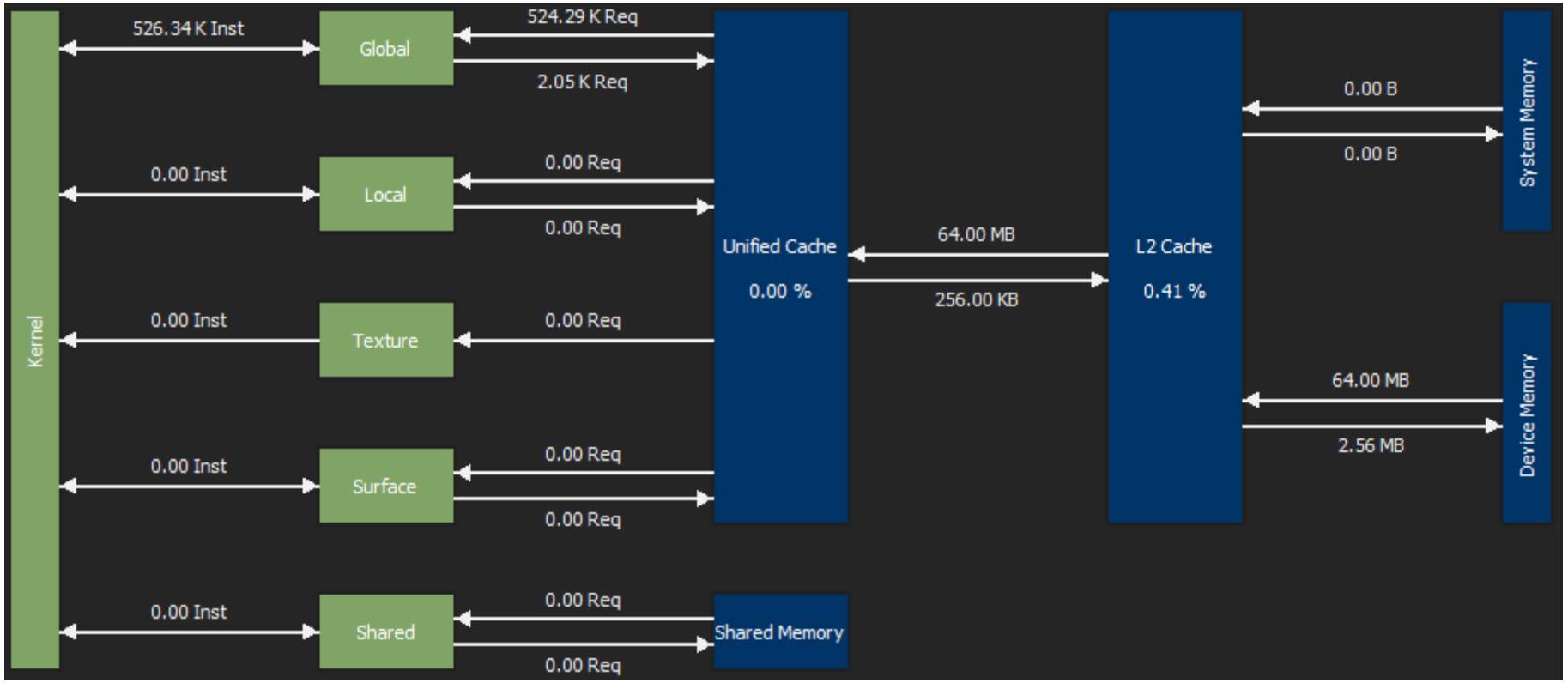
```
uchar4 in = tex2D(my_tex, colid, rowid);
```

RowAfterRow: Texture vs Global



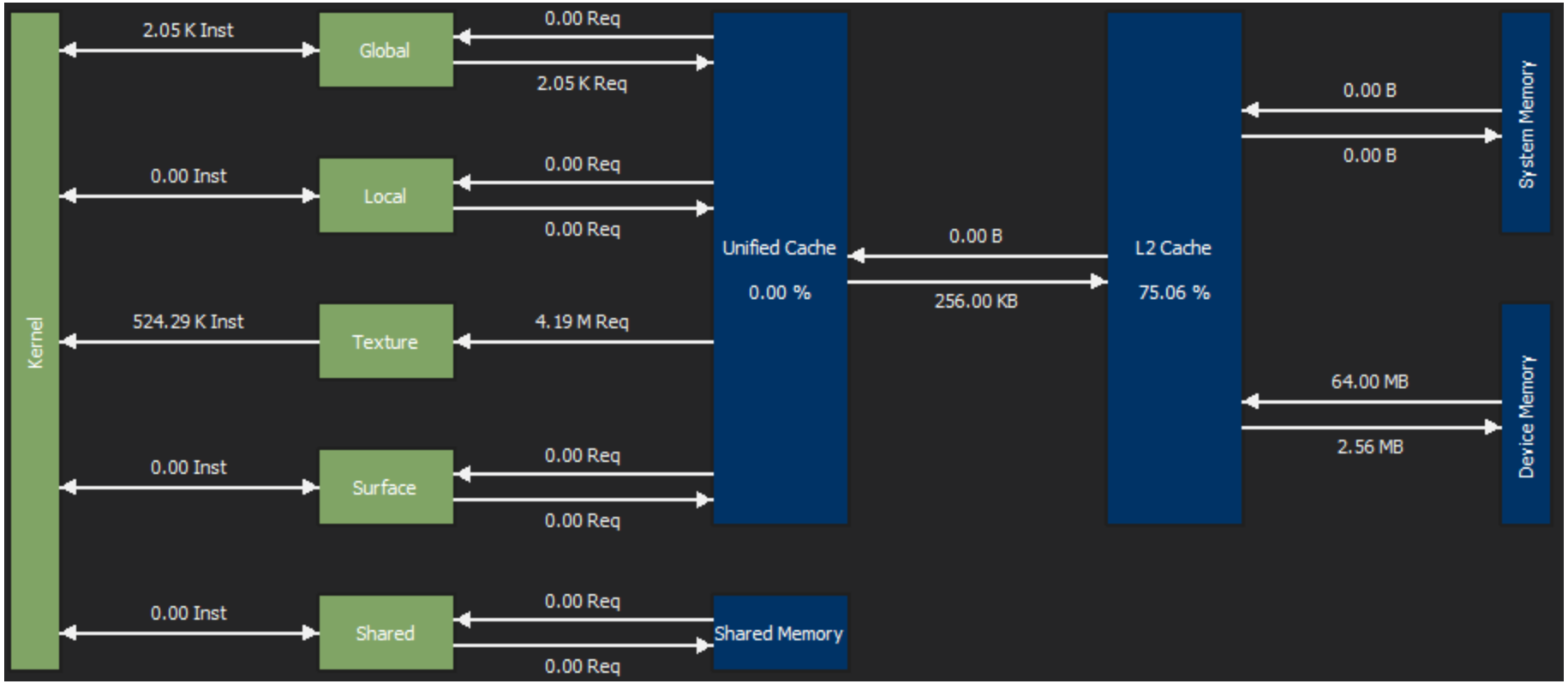
global linear (cudaMalloc) 445GB/s

RowAfterRow: Texture vs Global



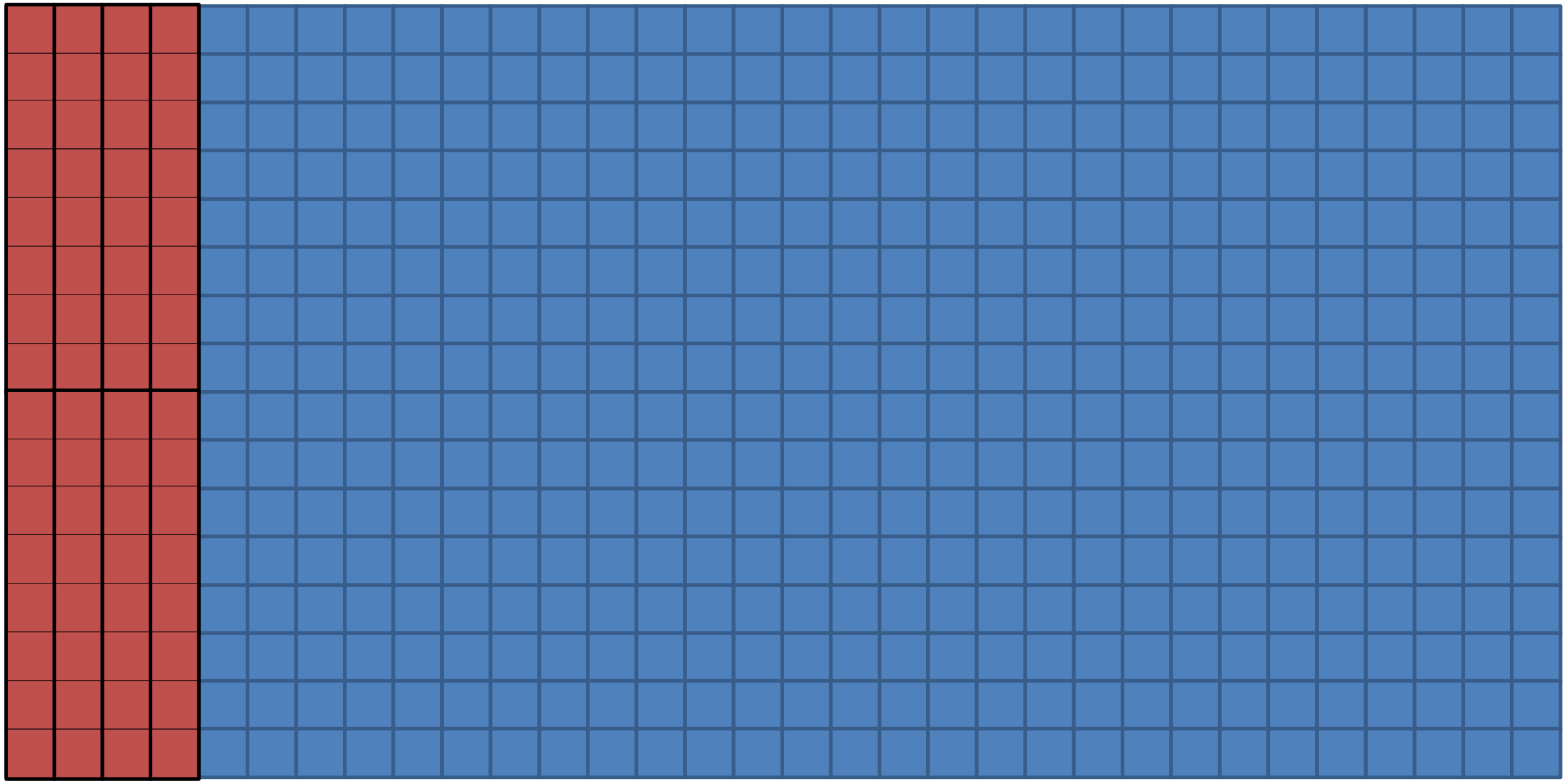
global restrict (cudaMalloc) 464GB/s

RowAfterRow: Texture vs Global

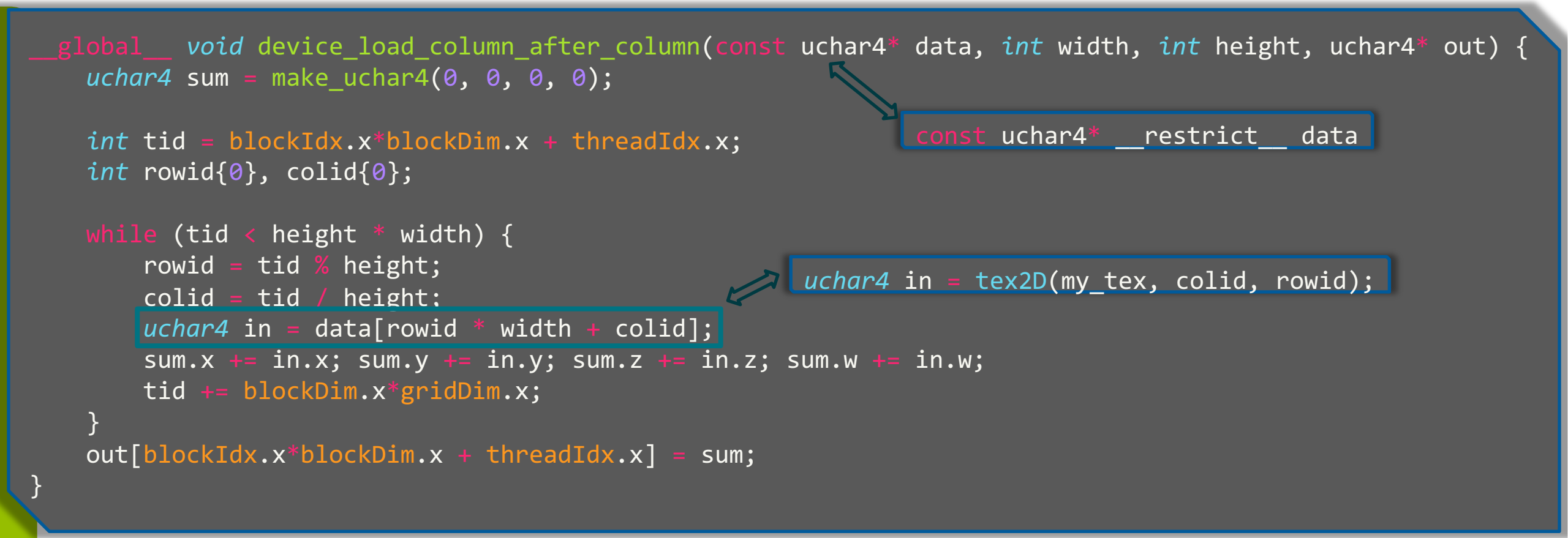


texture (cudaArray) 387GB/s

ColumnAfterColumn – 1D

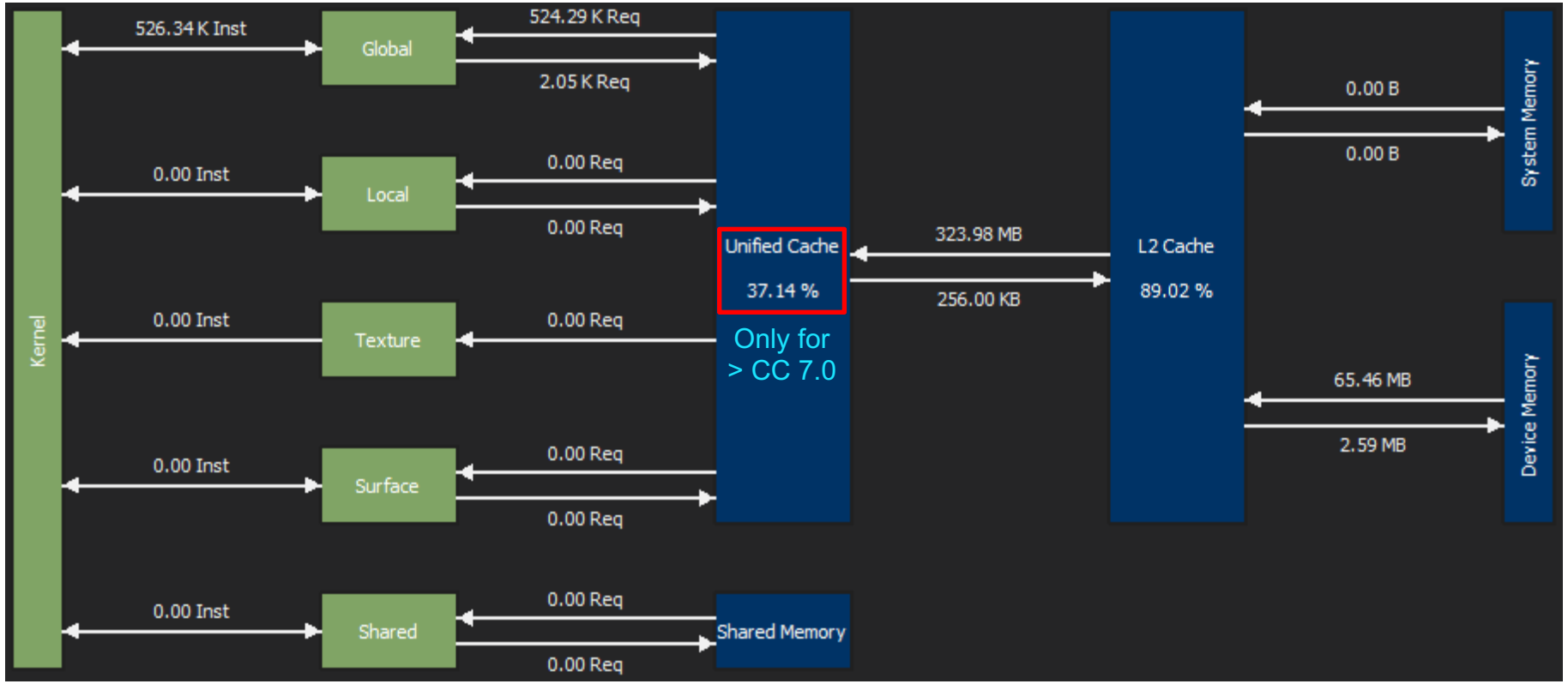


ColumnAfterColumn: Texture vs Global



```
__global__ void device_load_column_after_column(const uchar4* data, int width, int height, uchar4* out) {
    uchar4 sum = make_uchar4(0, 0, 0, 0);

    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int rowid{0}, colid{0};

    while (tid < height * width) {
        rowid = tid % height;
        colid = tid / height;
        uchar4 in = data[rowid * width + colid];
        sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
        tid += blockDim.x*gridDim.x;
    }
    out[blockIdx.x*blockDim.x + threadIdx.x] = sum;
}
```

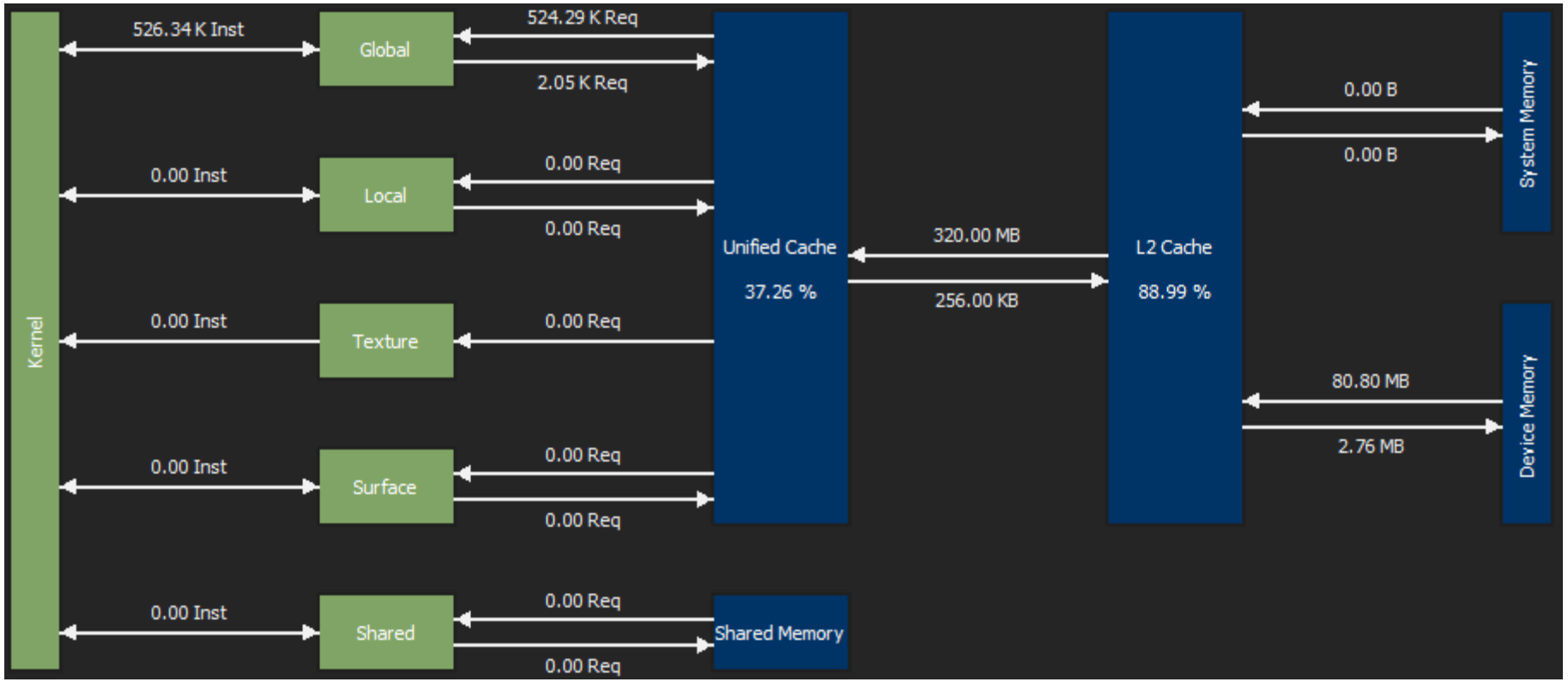
ColumnAfterColumn: Texture vs Global



global linear (cudaMalloc) 268GB/s

Was 445GB/s

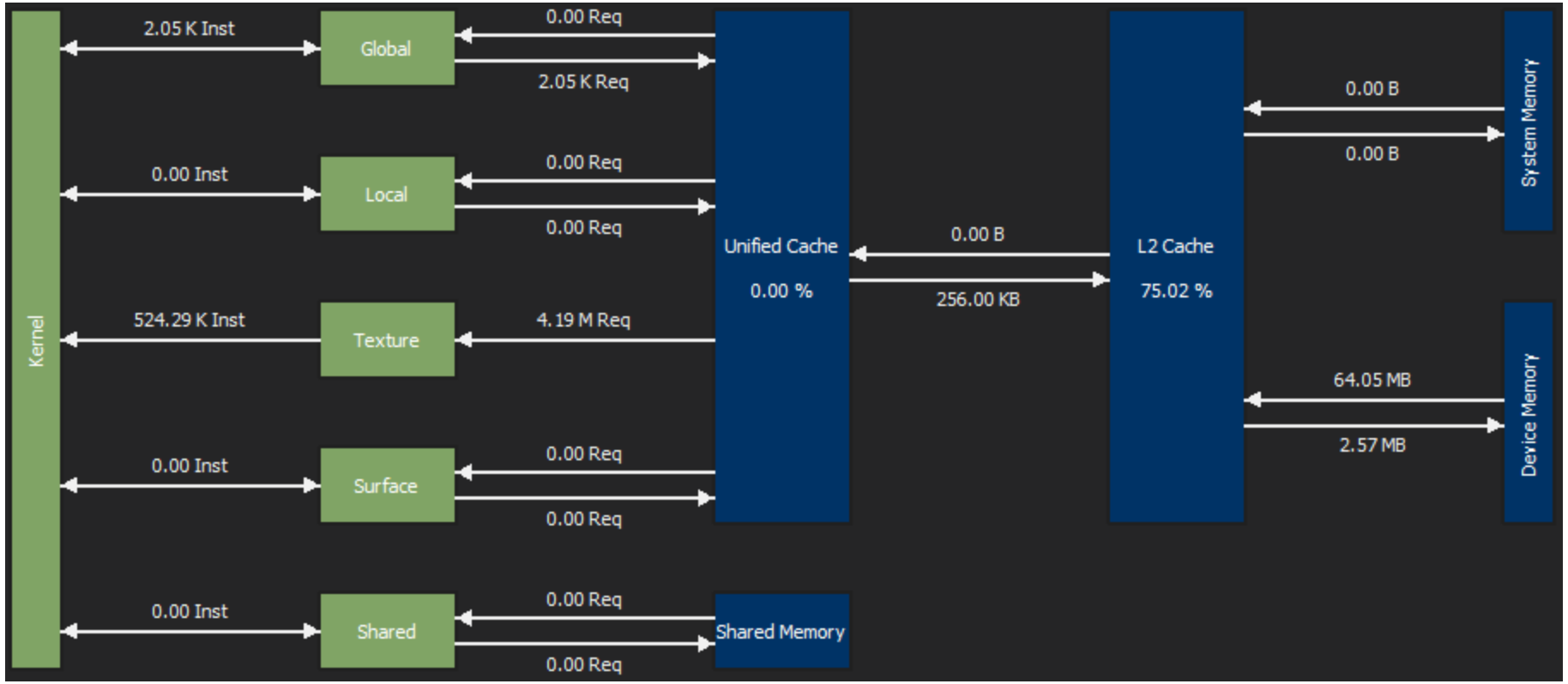
ColumnAfterColumn: Texture vs Global



global restrict (cudaMalloc) 270GB/s

Was 464GB/s

ColumnAfterColumn: Texture vs Global



texture (cudaArray) 368GB/s

Was 387GB/s

Texture vs Global: 1D Access

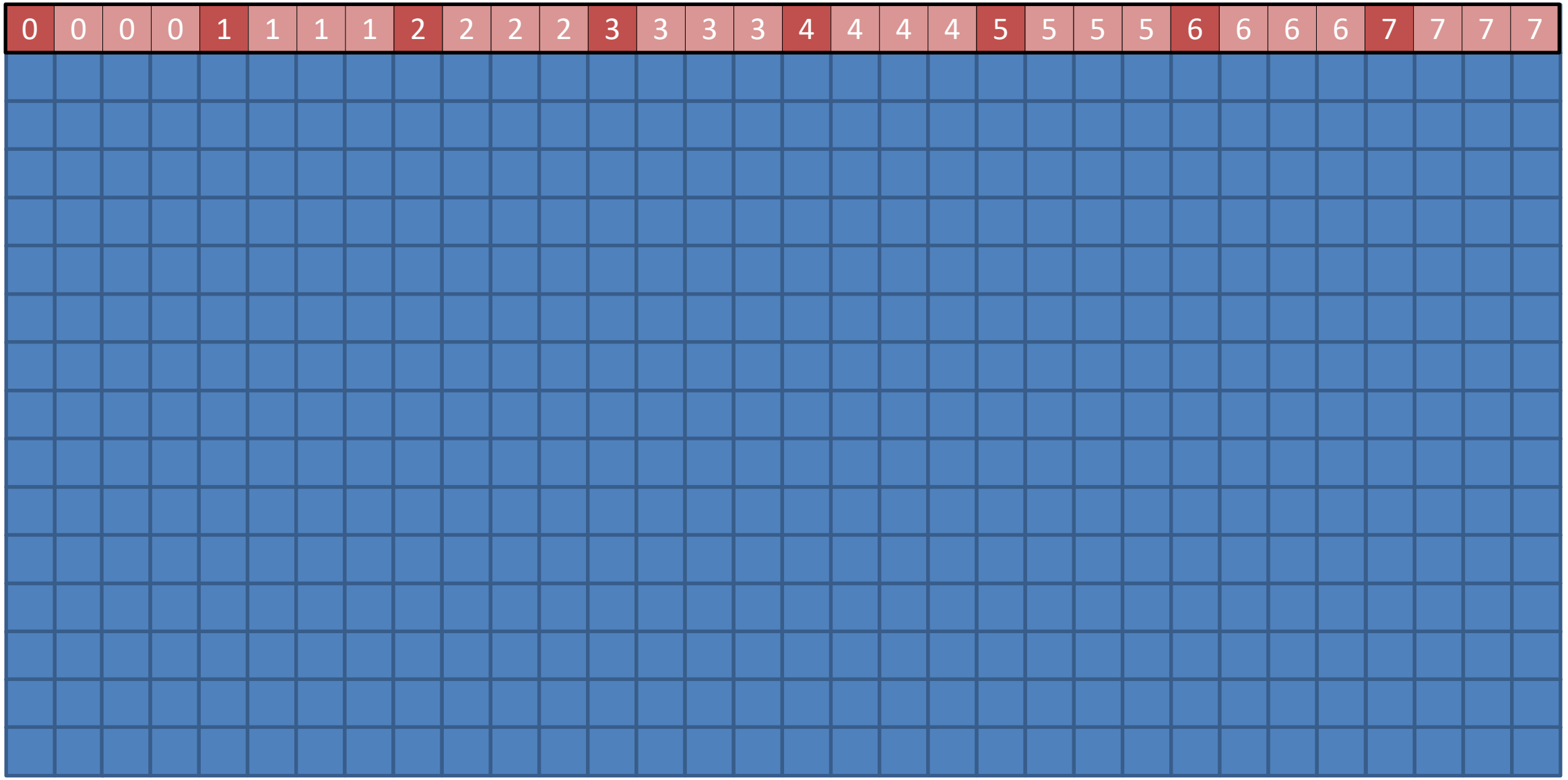
	ms				GB/s		
	Row						
Approach	1080 Ti	Titan V	2080 Ti		1080 Ti	Titan V	2080 Ti
Const Array	0,2499	0,1505	0,1368		268,6	445,9	490,6
Const Array Restrict	0,2468	0,1444	0,1384		271,9	464,8	484,9
Texture	0,345	0,1731	0,1679		194,5	387,8	399,8
	Column						
Approach	1080 Ti	Titan V	2080 Ti		1080 Ti	Titan V	2080 Ti
Const Array	1,032	0,2498	1,579		65,02	268,6	42,51
Const Array Restrict	0,964	0,2478	1,567		69,57	270,8	42,83
Texture	0,469	0,1823	0,1996		143	368,2	336,1

- Row access is similarly efficient with all type → virtually no cache usage
  - Slightly better with linear memory layout compared to textures
- Column access significantly slower for global vs. texture
  - Texture slightly slower than row access

RowAfterRow – 1D Cache <2>

0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7

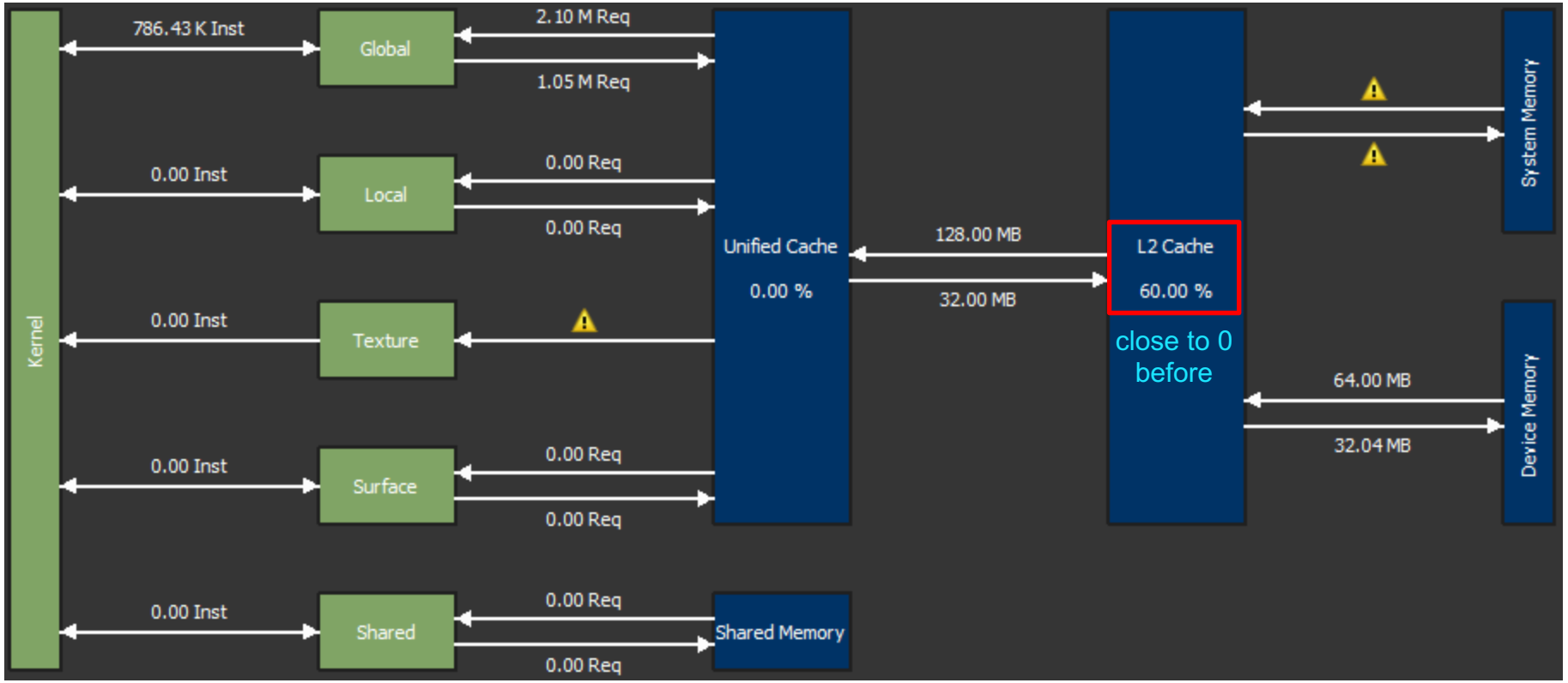
RowAfterRow – 1D Cache <4>



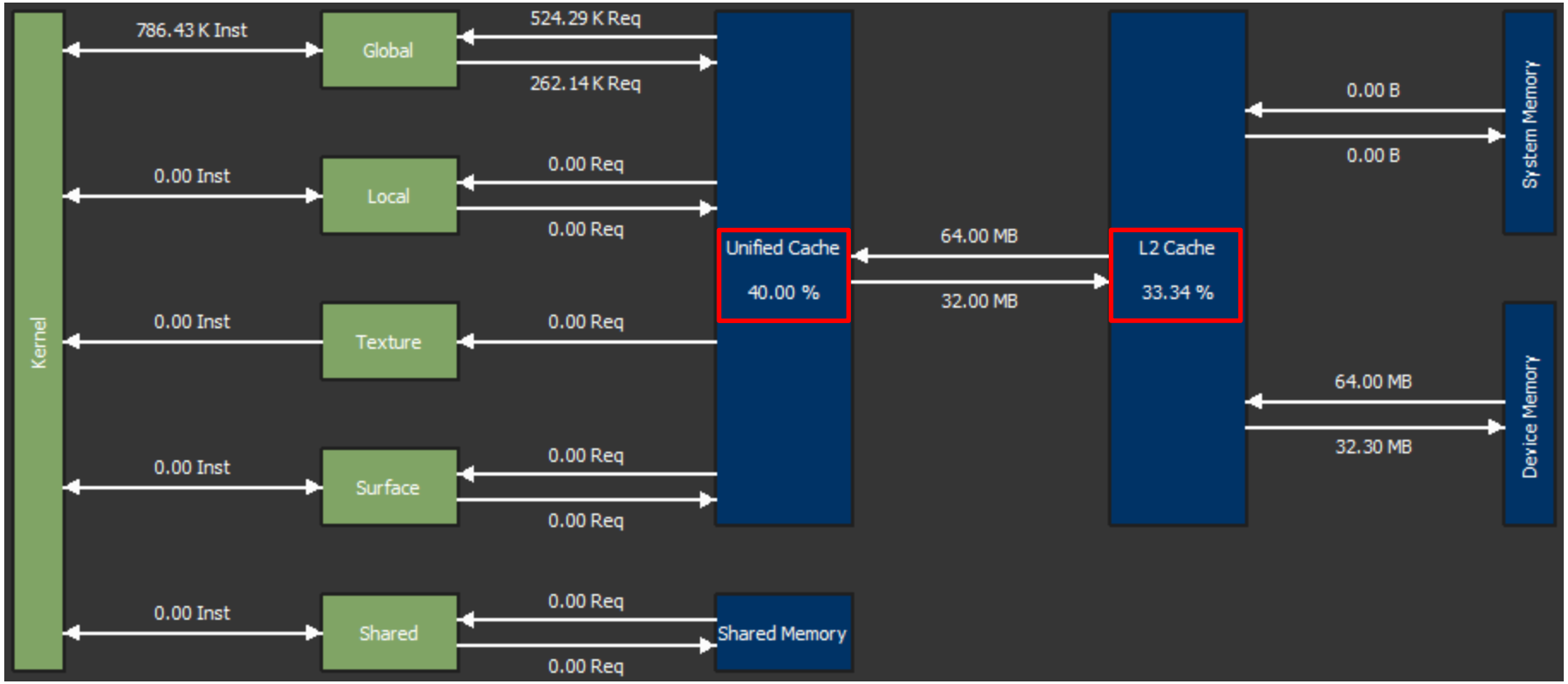
RowAfterRow: 1D Cache

```
template <int CACHE_OFFSET>
__global__ void device_load_row_after_row_offset(const uchar4* data, int width, int height, uchar4* out) {
    uchar4 sum = make_uchar4(0, 0, 0, 0);

    int tid = (blockIdx.x*blockDim.x + threadIdx.x) * CACHE_OFFSET;
    int rowid{ 0 }, colid{ 0 };

    #pragma unroll
    for (int i = 0; i < CACHE_OFFSET; ++i, ++tid) {
        rowid = tid / width;
        colid = tid % width;
        uchar4 in = data[rowid * width + colid];
        sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
    }
    out[blockIdx.x*blockDim.x + threadIdx.x] = sum;
}
```

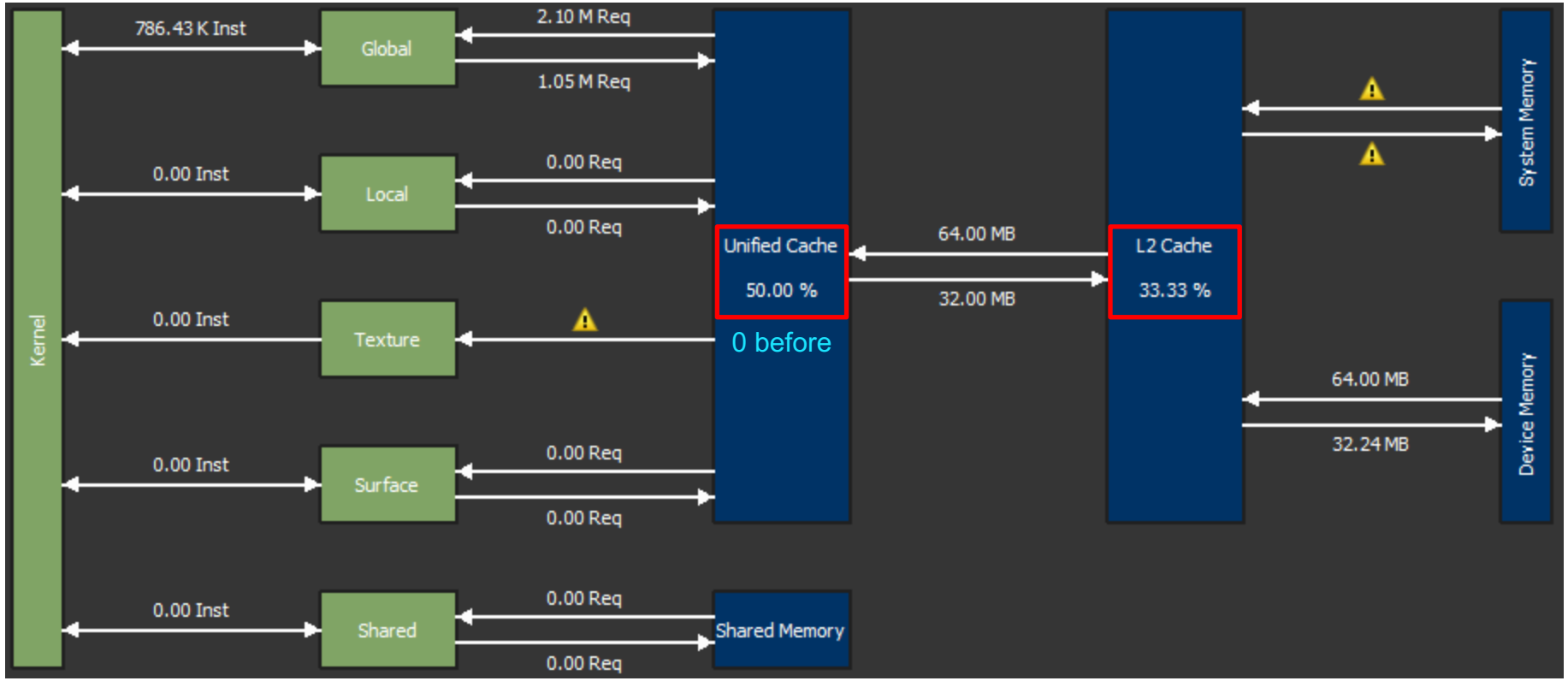
Const : 1D Cache <2> GTX 1080 Ti



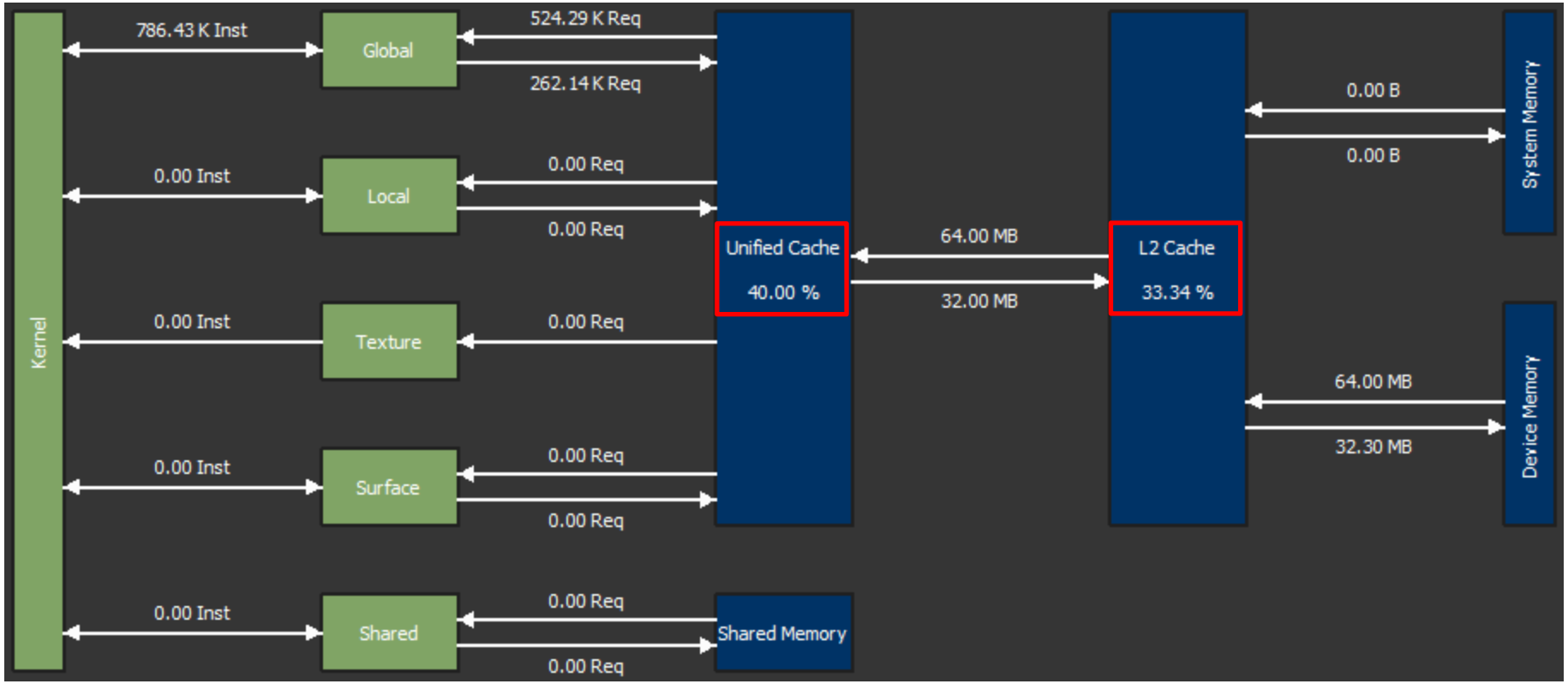
Const : 1D Cache <2> Titan V



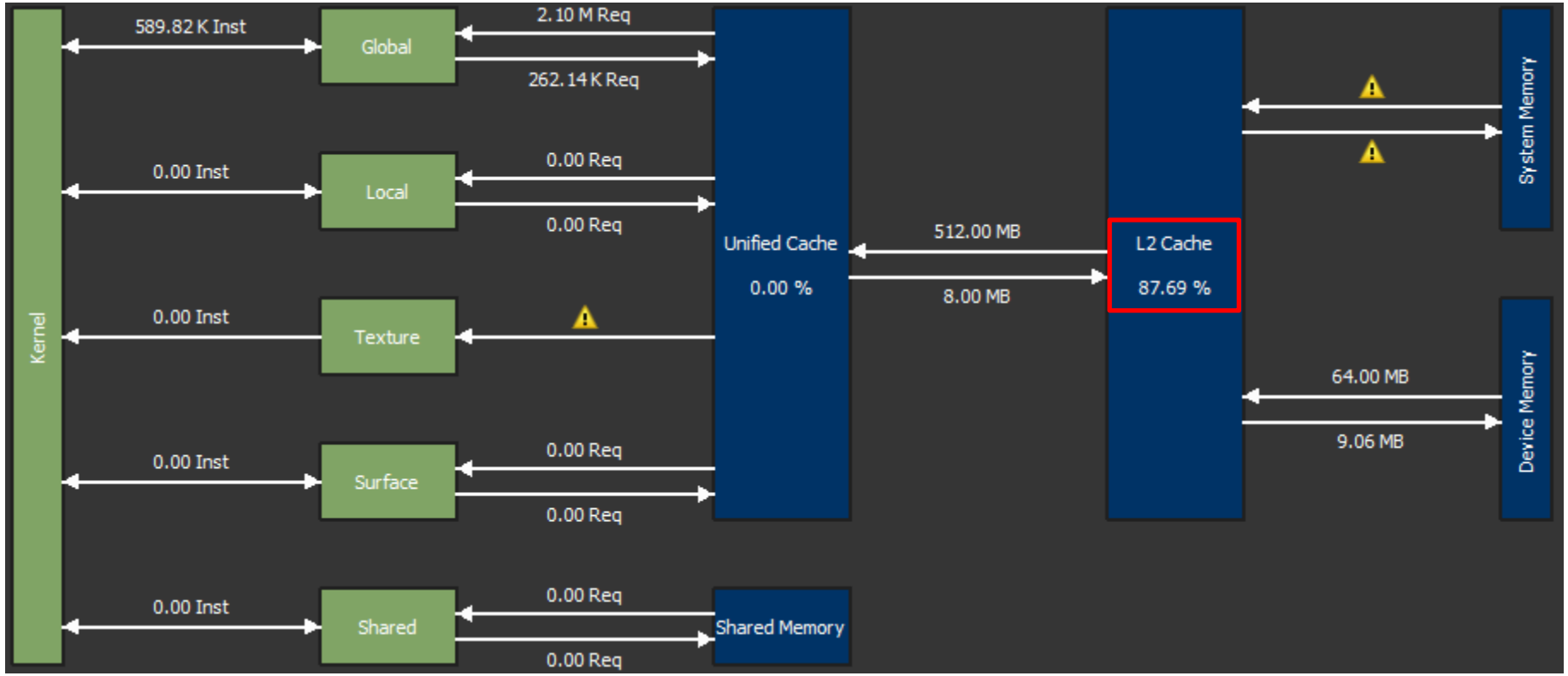
Restrict : 1D Cache <2> GTX 1080 Ti



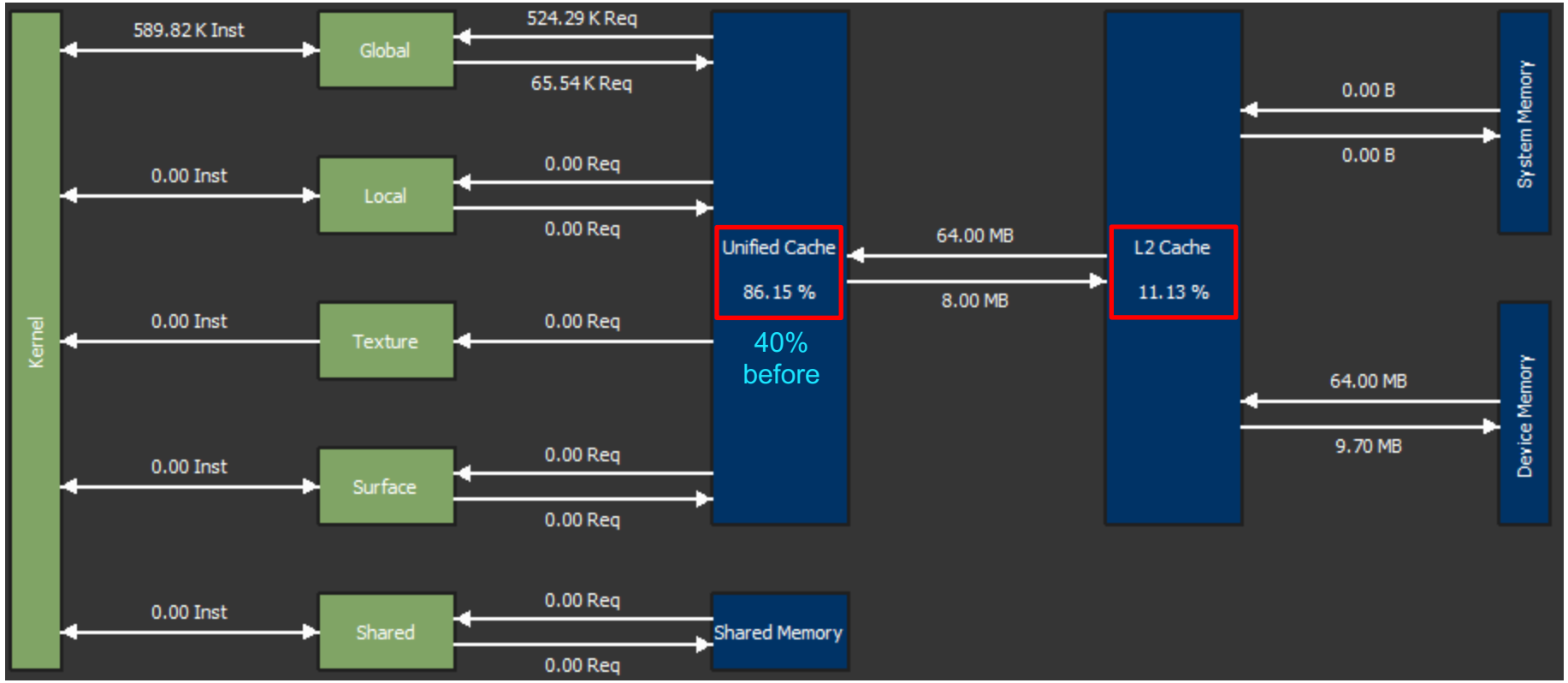
Restrict : 1D Cache <2> Titan V



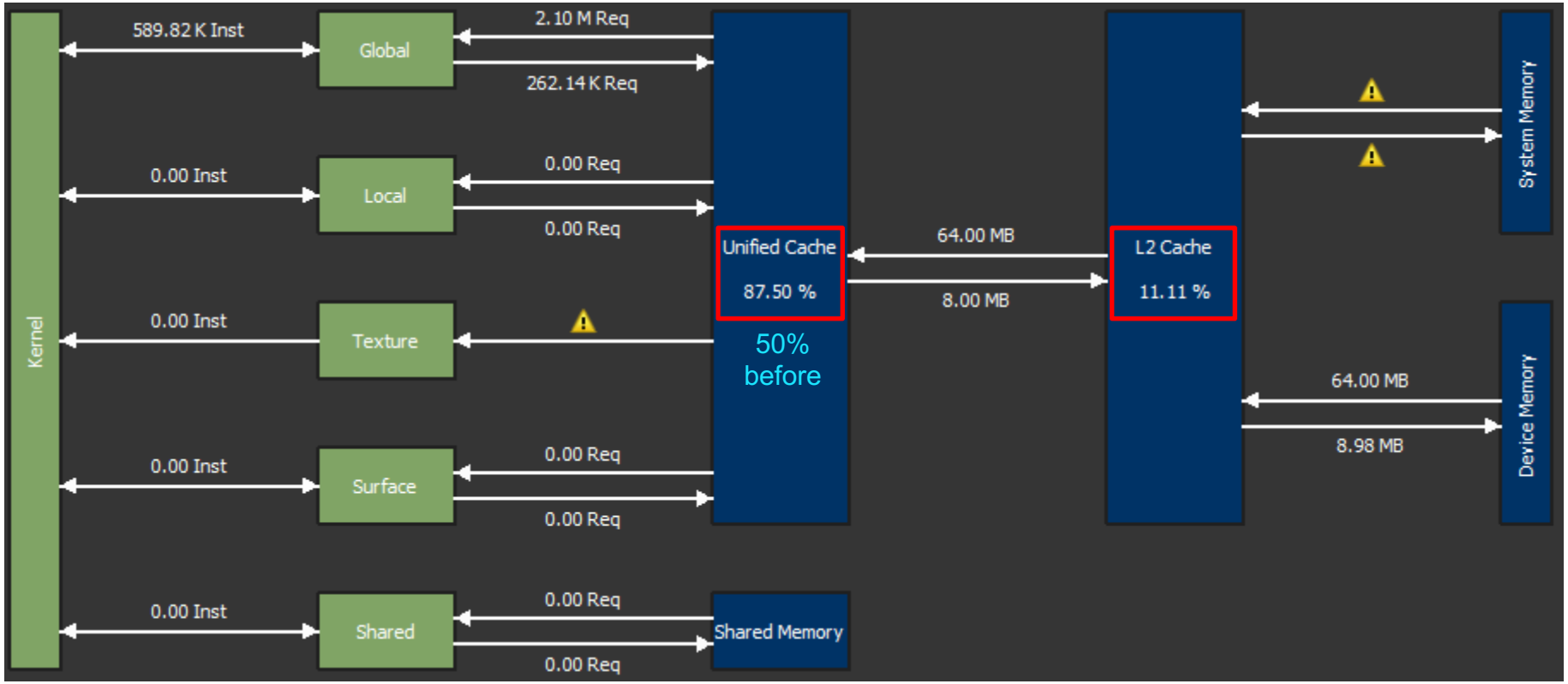
Const : 1D Cache <8> GTX 1080 Ti



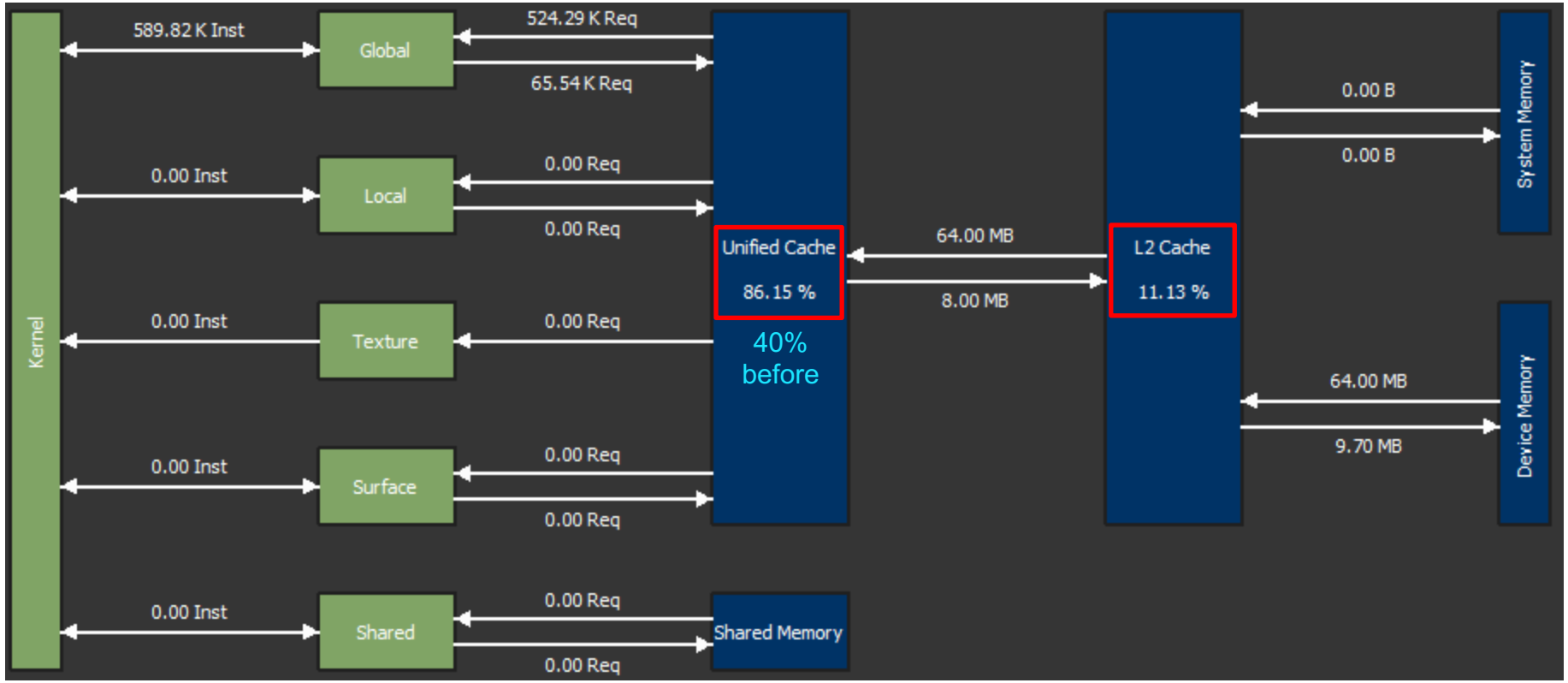
Const : 1D Cache <8> Titan V



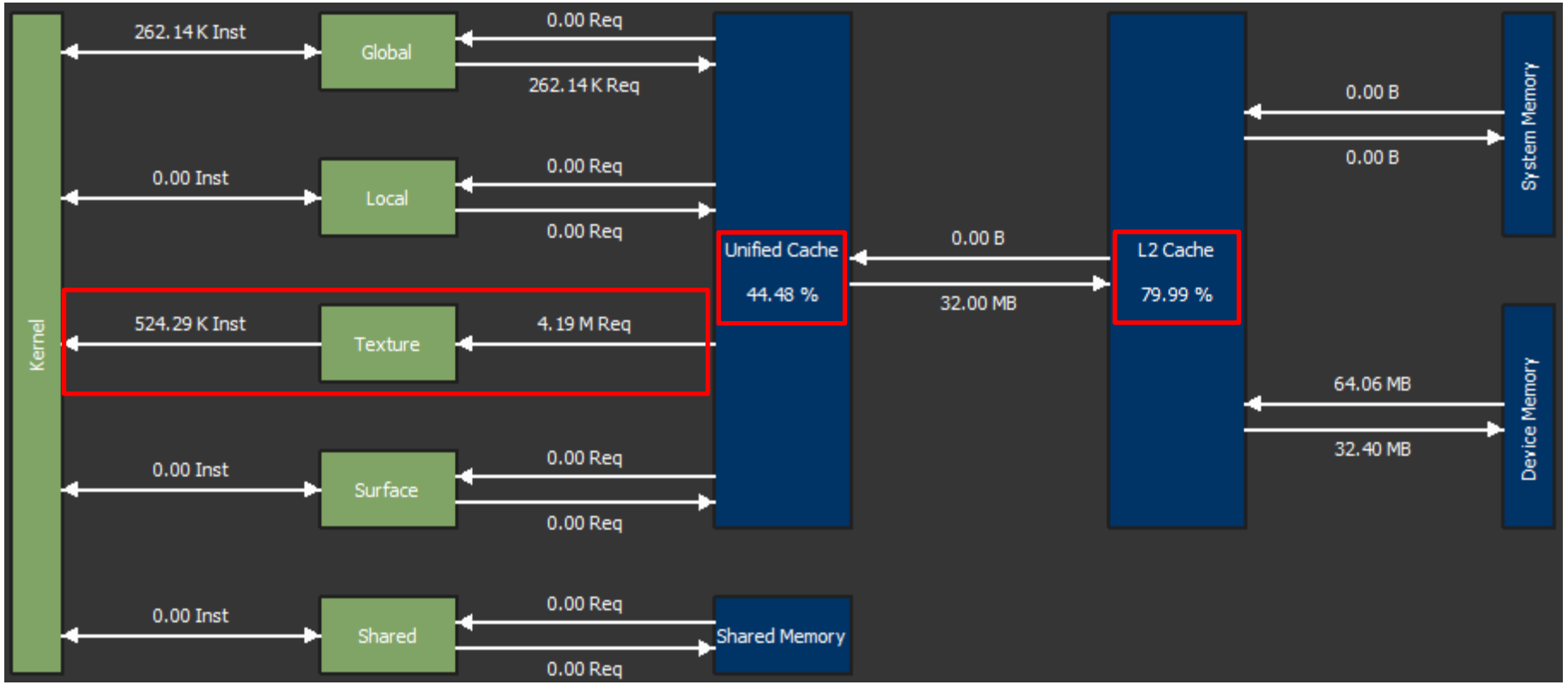
Restrict : 1D Cache <8> GTX 1080 Ti



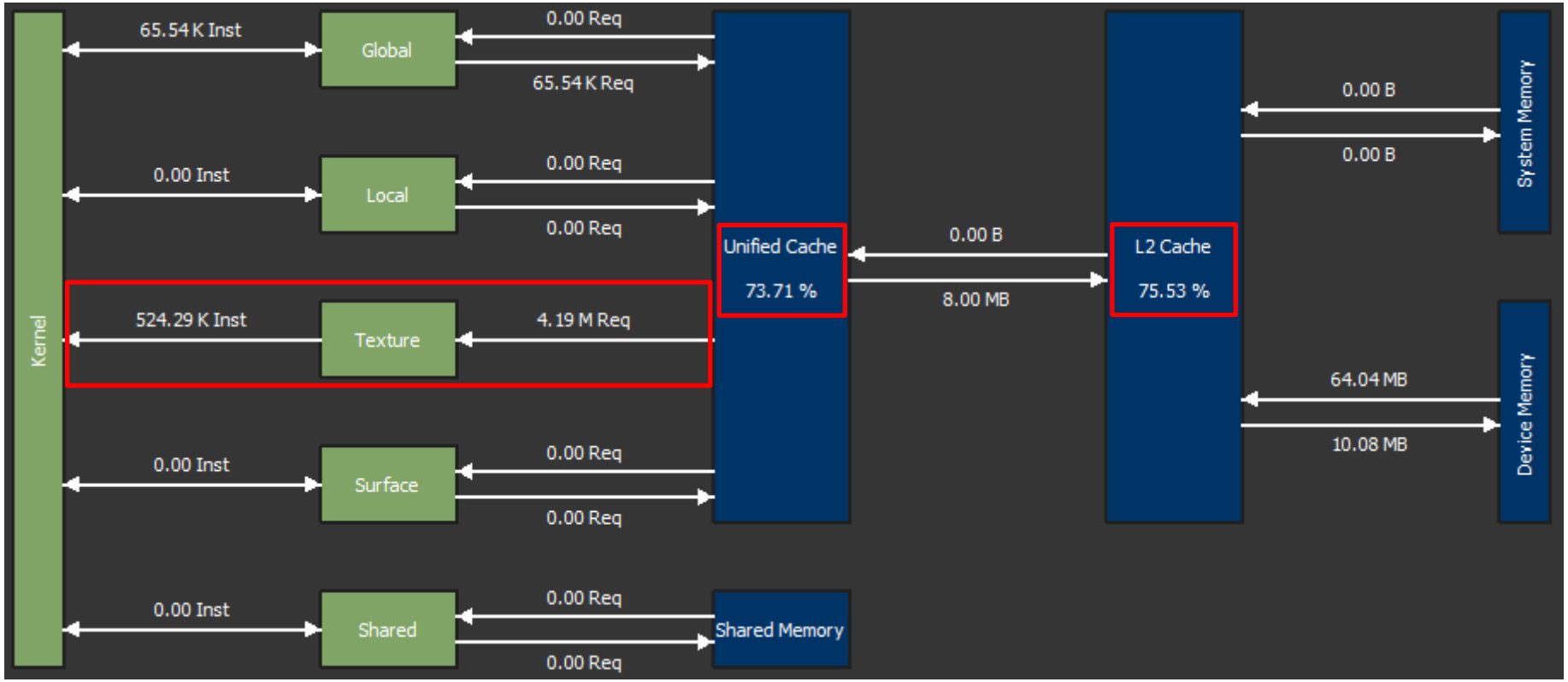
Restrict : 1D Cache <8> Titan V



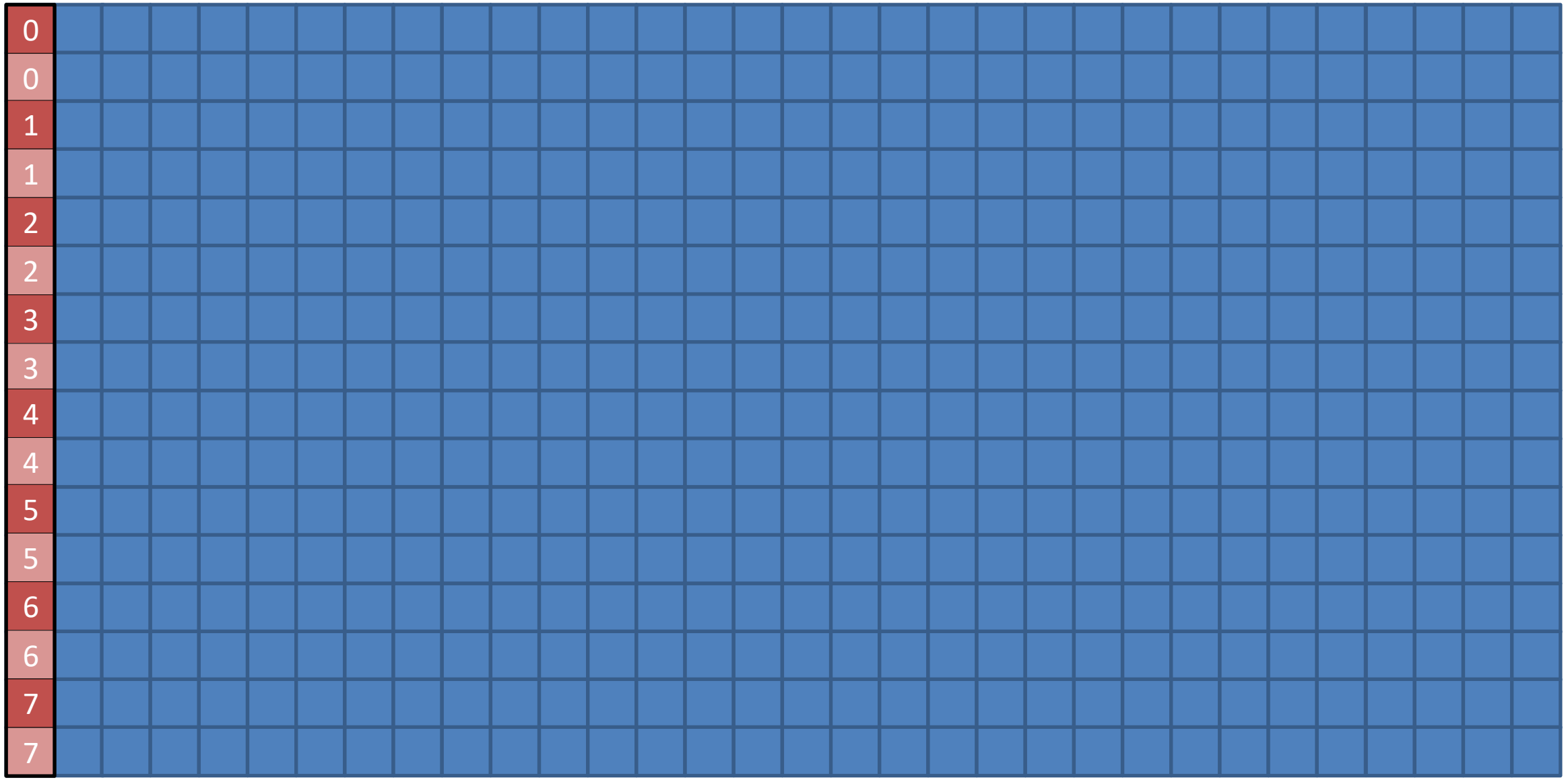
Texture : 1D Cache <2> Titan V



Texture : 1D Cache <8> Titan V



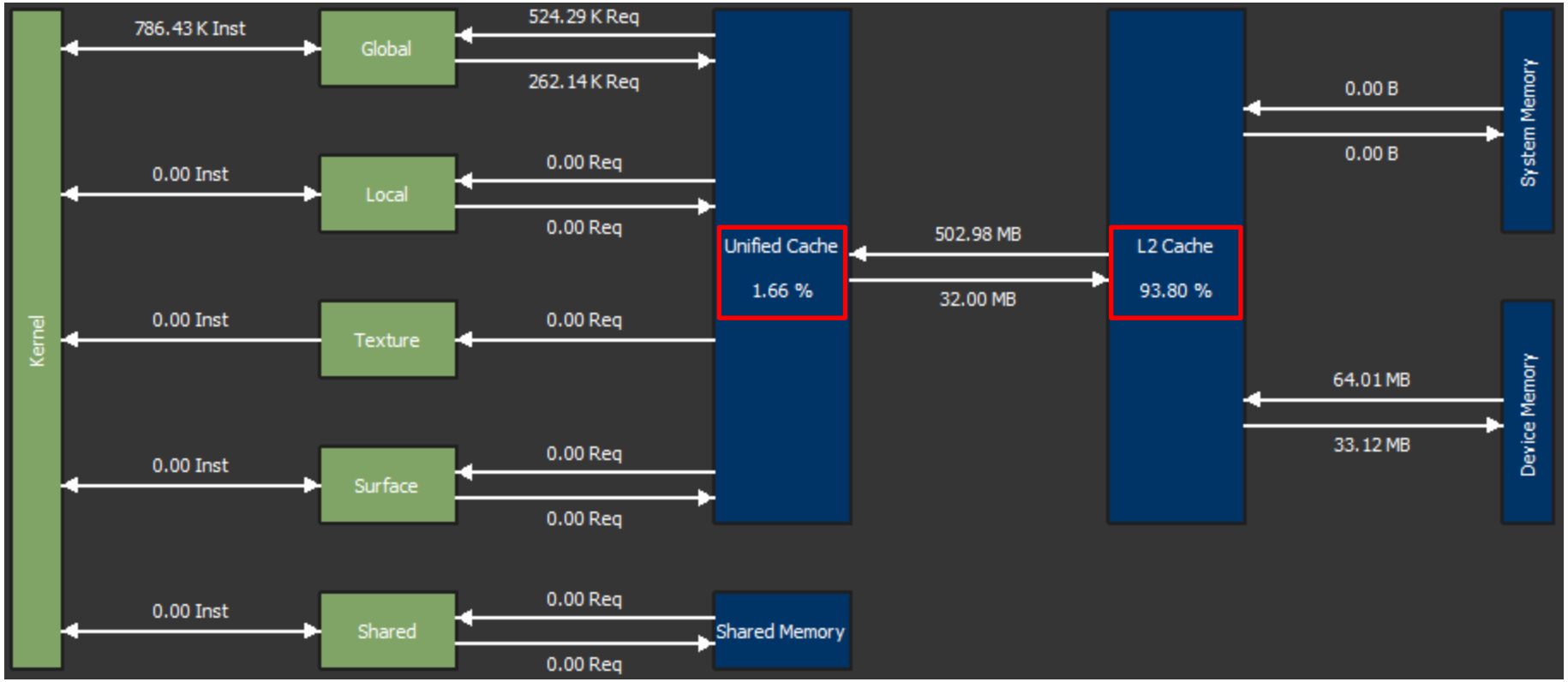
ColumnAfterColumn – 1D Cache



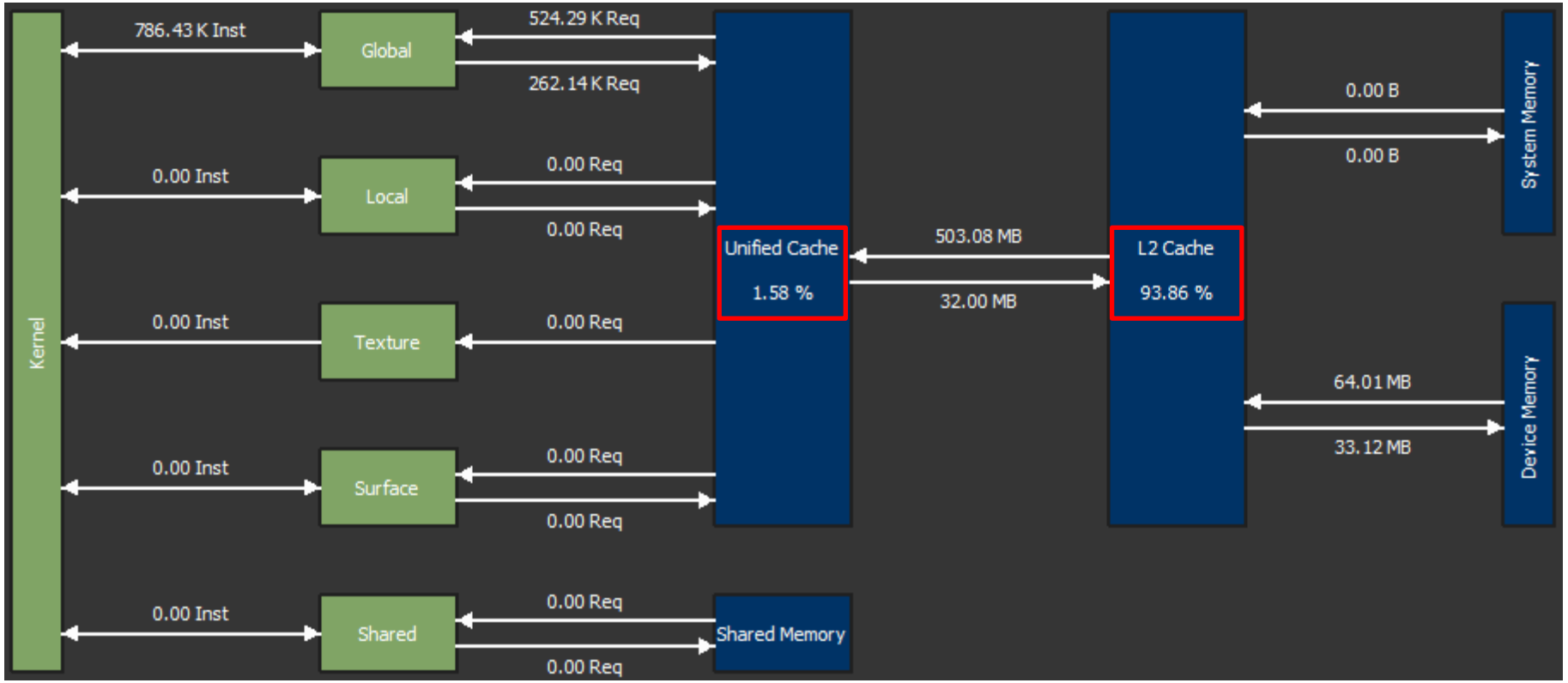
ColumnAfterColumn: 1D Cache

```
template <int CACHE_OFFSET>
__global__ void device_load_col_after_col_offset(const uchar4* data, int width, int height, uchar4* out) {
    uchar4 sum = make_uchar4(0, 0, 0, 0);

    int tid = (blockIdx.x*blockDim.x + threadIdx.x) * CACHE_OFFSET;
    int rowid{ 0 }, colid{ 0 };

    #pragma unroll
    for (int i = 0; i < CACHE_OFFSET; ++i, ++tid) {
        rowid = tid % height;
        colid = tid / height;
        uchar4 in = data[rowid * width + colid];
        sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
    }
    out[blockIdx.x*blockDim.x + threadIdx.x] = sum;
}
```

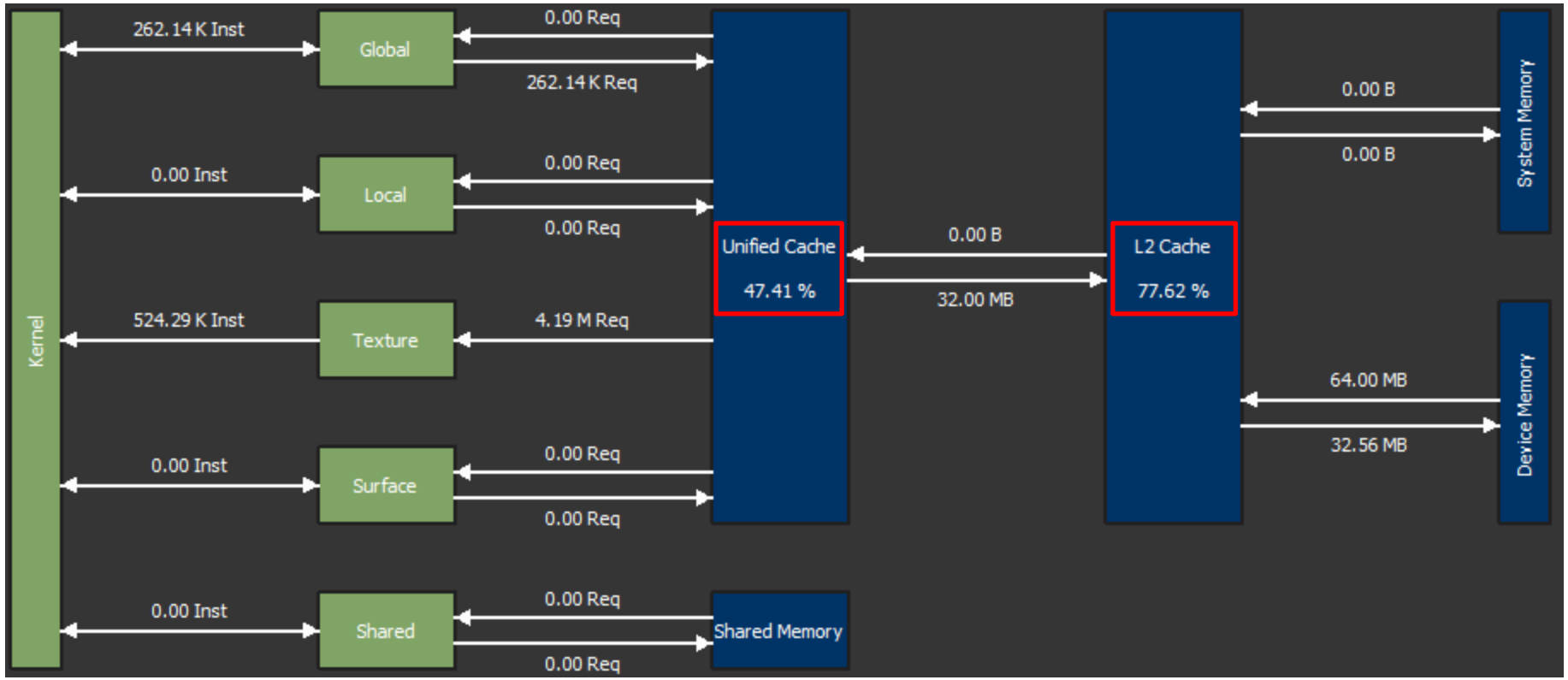
Const : 1D Cache <2> Titan V



Restrict : 1D Cache <2> Titan V



Texture : 1D Cache <2> Titan V



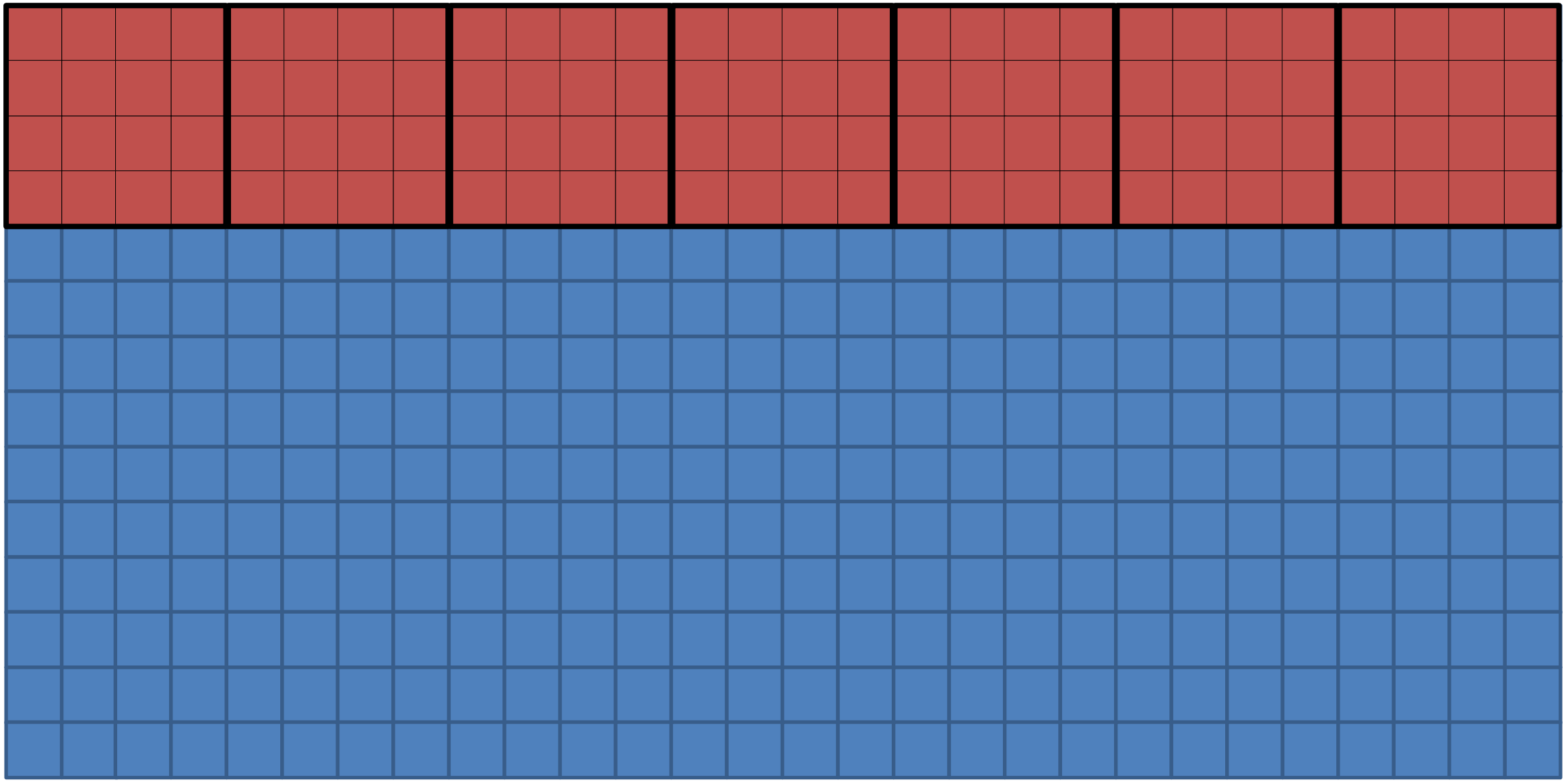
1D Cache – Row

	ms				GB/s		
	Cache <2>						
Approach	1080 Ti	Titan V	2080 Ti		1080 Ti	Titan V	2080 Ti
Const Array	0,2772	0,1669	0,1800		242,1	402,1	372,9
Const Array Restrict	0,2898	0,1731	0,1844		231,6	387,8	364,0
Texture	0,2827	0,1751	0,1865		237,4	383,3	359,8
	Cache <4>						
Approach	1080 Ti	Titan V	2080 Ti		1080 Ti	Titan V	2080 Ti
Const Array	0,2829	0,1475	0,1558		237,2	455,1	430,7
Const Array Restrict	0,2374	0,1464	0,1556		282,7	458,3	431,2
Texture	0,2593	0,1526	0,1575		258,8	439,8	426,2
	Cache <8>						
Approach	1080 Ti	Titan V	2080 Ti		1080 Ti	Titan V	2080 Ti
Const Array	0,5497	0,1331	0,1457		122,1	504,1	460,5
Const Array Restrict	0,2170	0,1342	0,1454		309,3	500,2	461,4
Texture	0,3922	0,1853	0,1971		171,1	362,1	340,6

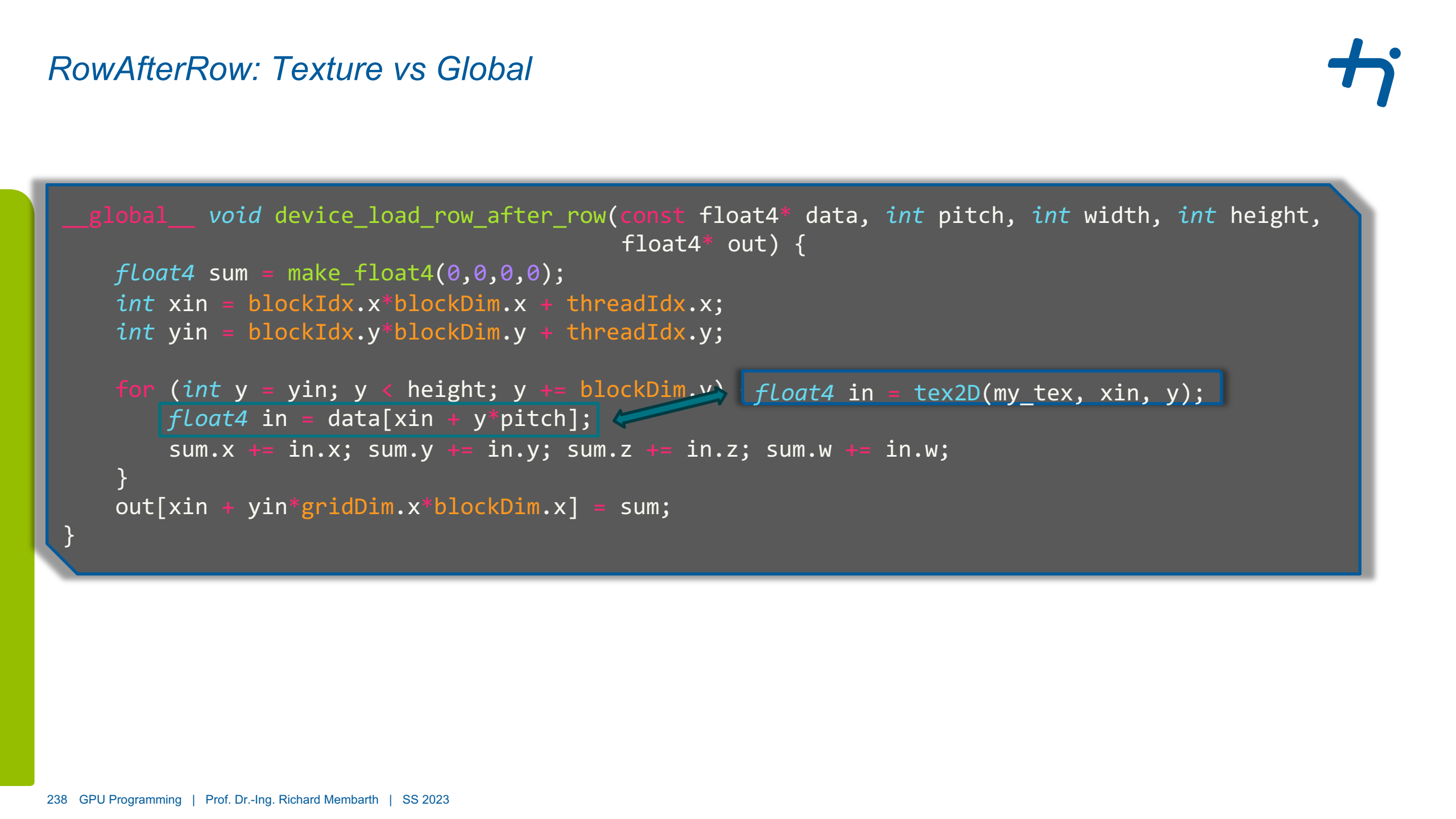
1D Cache – Column

	ms				GB/s		
	Cache <2>						
Approach	1080 Ti	Titan V	2080 Ti		1080 Ti	Titan V	2080 Ti
Const Array	0,9069	0,3839	0,4526		73,99	174,8	148,3
Const Array Restrict	0,8982	0,3830	0,4510		74,72	175,2	148,8
Texture	0,2977	0,1772	0,2069		225,4	387,8	324,3
	Cache <4>						
Approach	1080 Ti	Titan V	2080 Ti		1080 Ti	Titan V	2080 Ti
Const Array	0,9010	0,3758	0,4154		74,49	178,6	161,6
Const Array Restrict	0,8980	0,3758	0,4135		74,74	178,6	162,3
Texture	0,3859	0,1997	0,1840		173,9	336,1	364,8
	Cache <8>						
Approach	1080 Ti	Titan V	2080 Ti		1080 Ti	Titan V	2080 Ti
Const Array	0,9358	0,5069	0,4832		71,71	132,4	138,9
Const Array Restrict	0,9337	0,5100	0,4791		71,87	131,6	140,1
Texture	0,4895	0,1956	0,2091		137,1	343,1	321,0

RowAfterRow – 2D



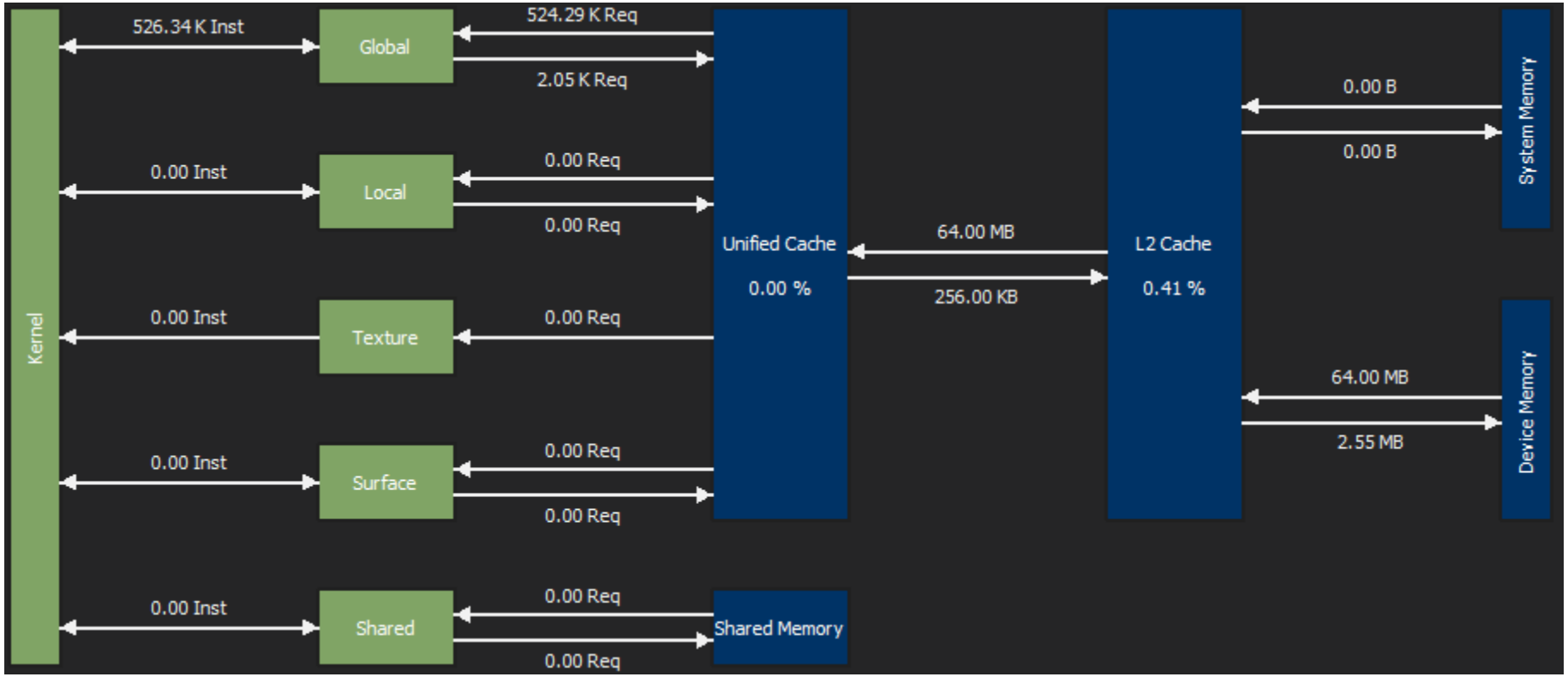
RowAfterRow: Texture vs Global



```
__global__ void device_load_row_after_row(const float4* data, int pitch, int width, int height,
                                          float4* out) {
    float4 sum = make_float4(0,0,0,0);
    int xin = blockIdx.x*blockDim.x + threadIdx.x;
    int yin = blockIdx.y*blockDim.y + threadIdx.y;

    for (int y = yin; y < height; y += blockDim.y) {
        float4 in = data[xin + y*pitch];
        sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
    }
    out[xin + yin*gridDim.x*blockDim.x] = sum;
}
```

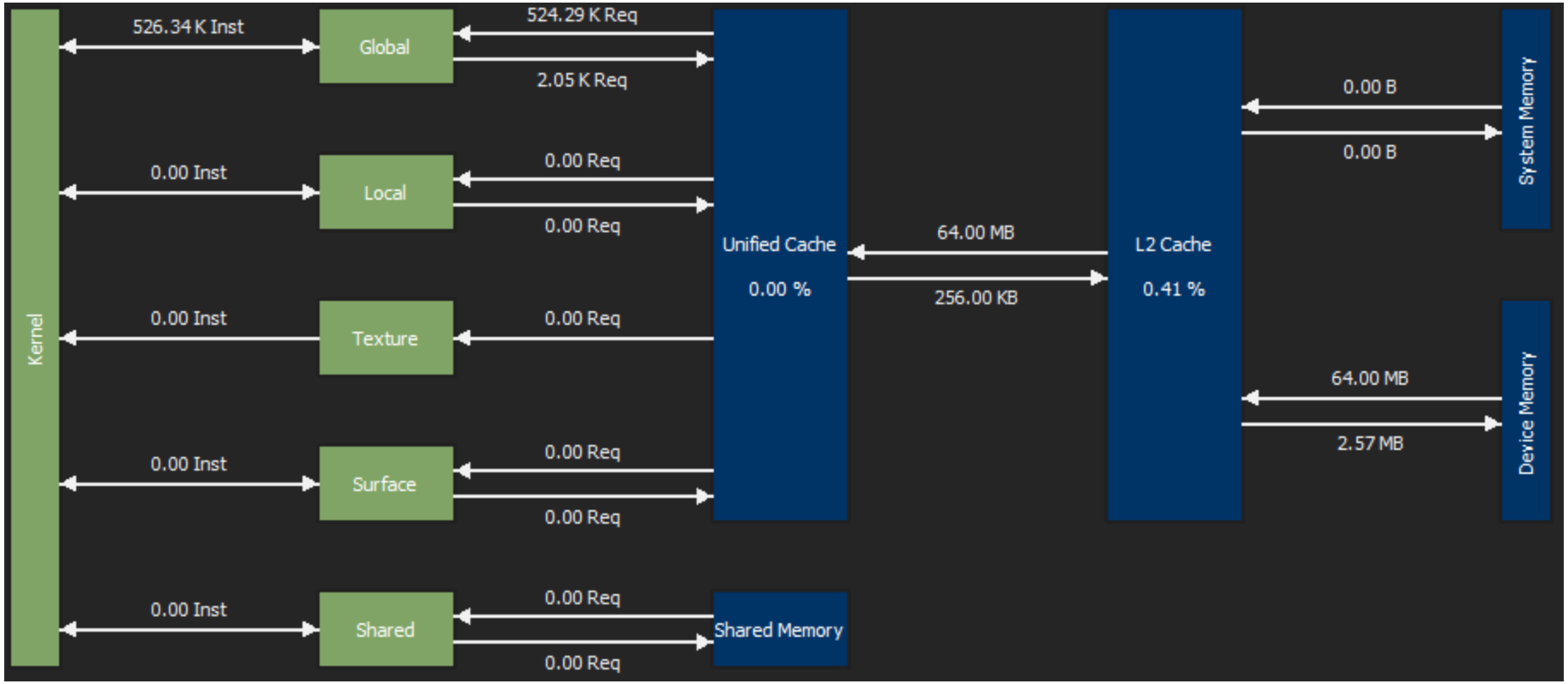
```
float4 in = tex2D(my_tex, xin, y);
```

RowAfterRow: Texture vs Global



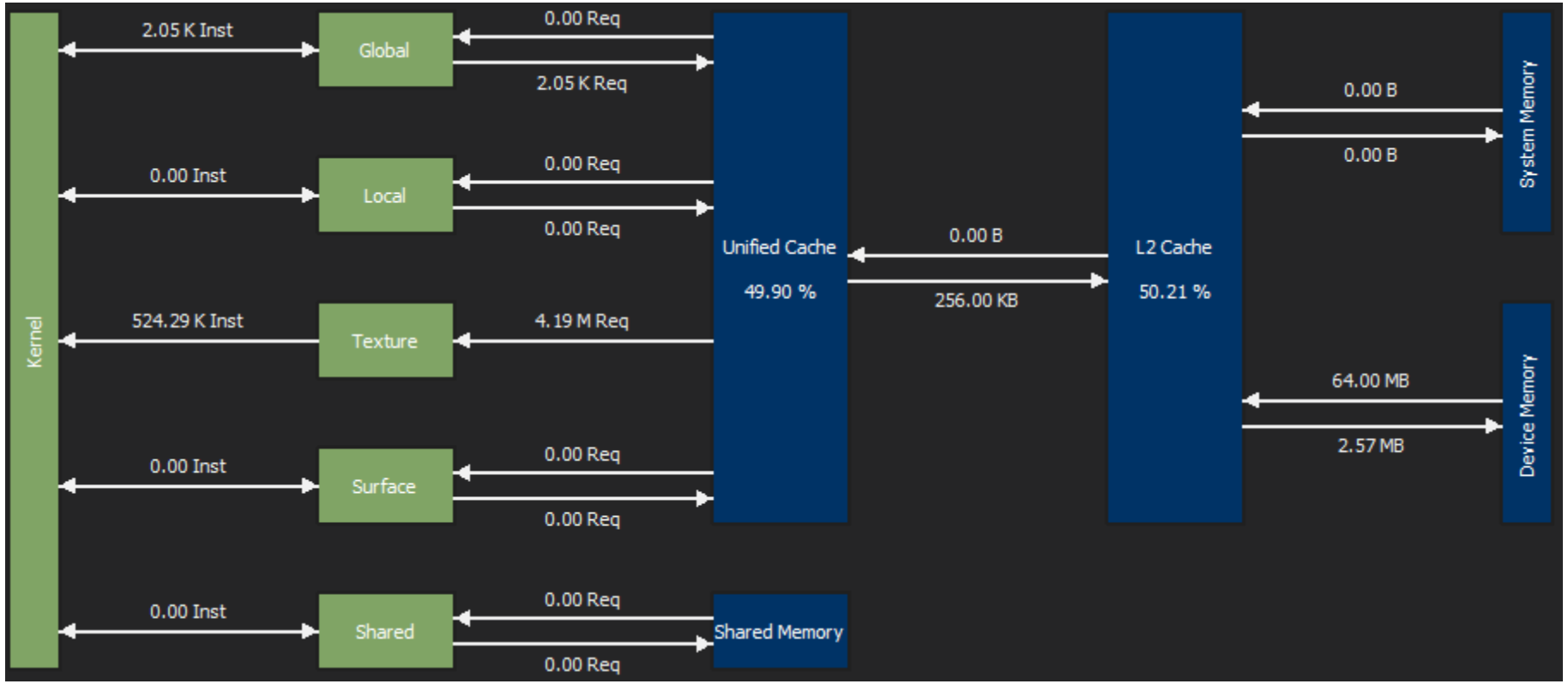
global linear (cudaMalloc) 461GB/s

RowAfterRow: Texture vs Global



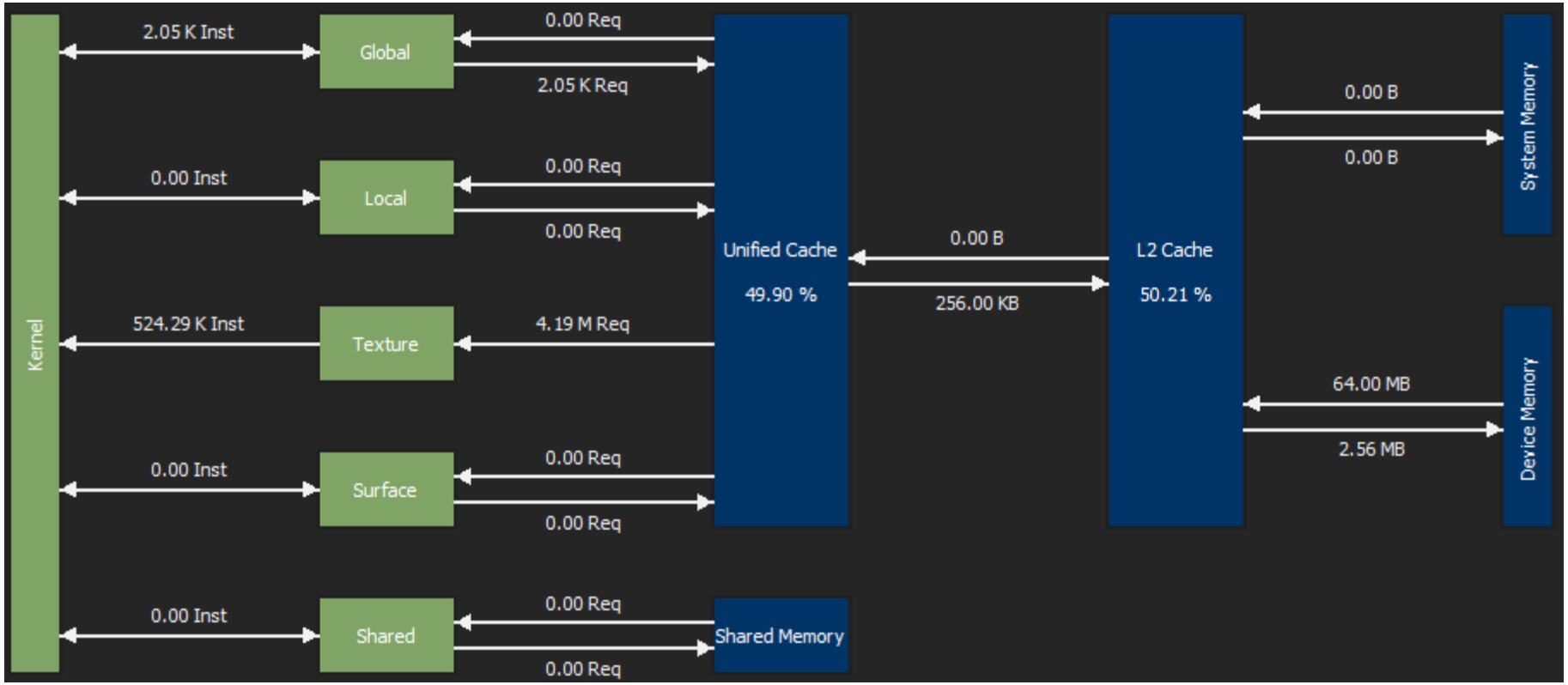
global 2D (cudaMallocPitch) 464GB/s

RowAfterRow: Texture vs Global



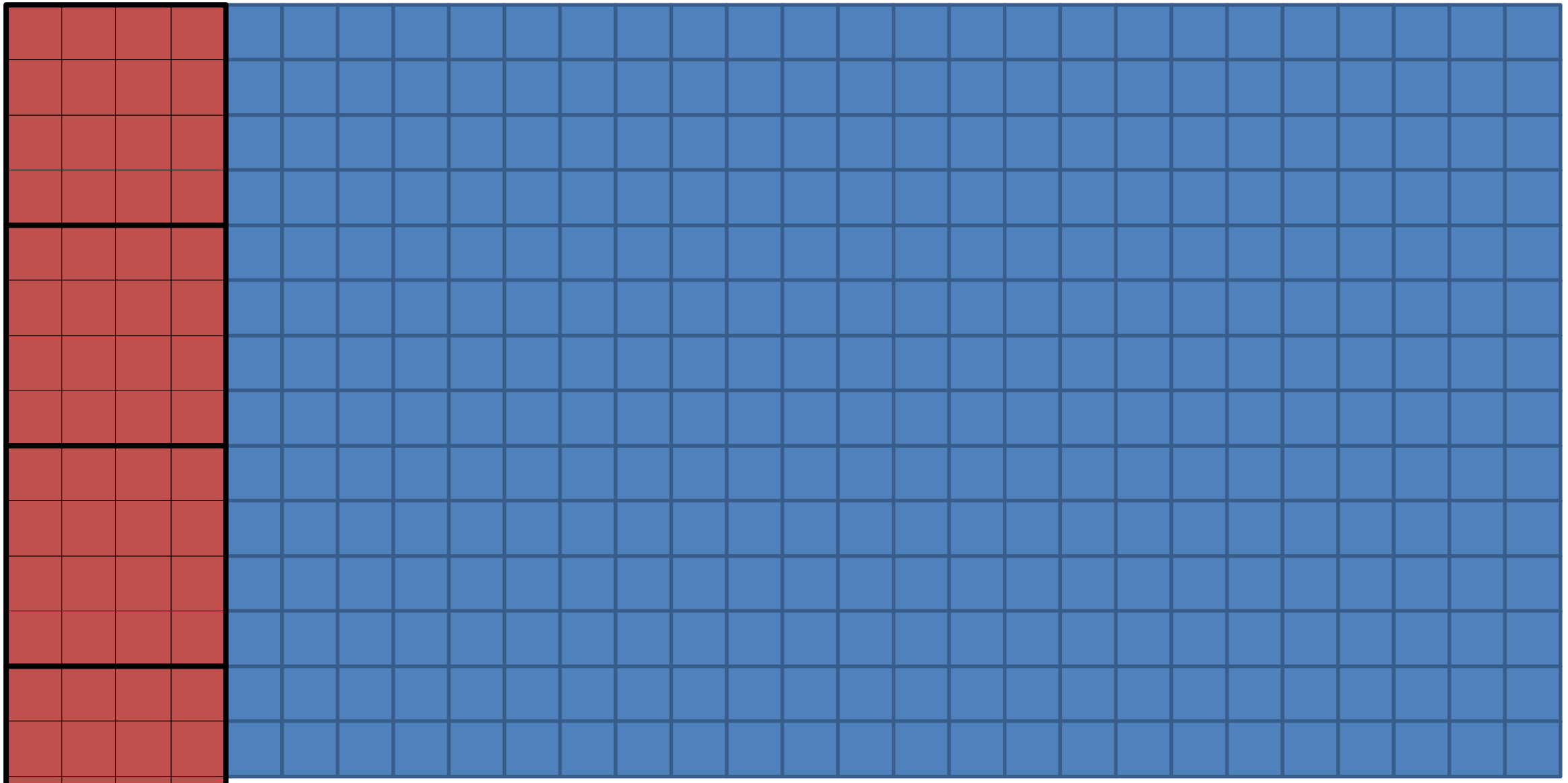
texture (cudaArray) 434GB/s

RowAfterRow: Texture vs Global



texture (cudaMallocPitch) 422GB/s

ColumnAfterColumn

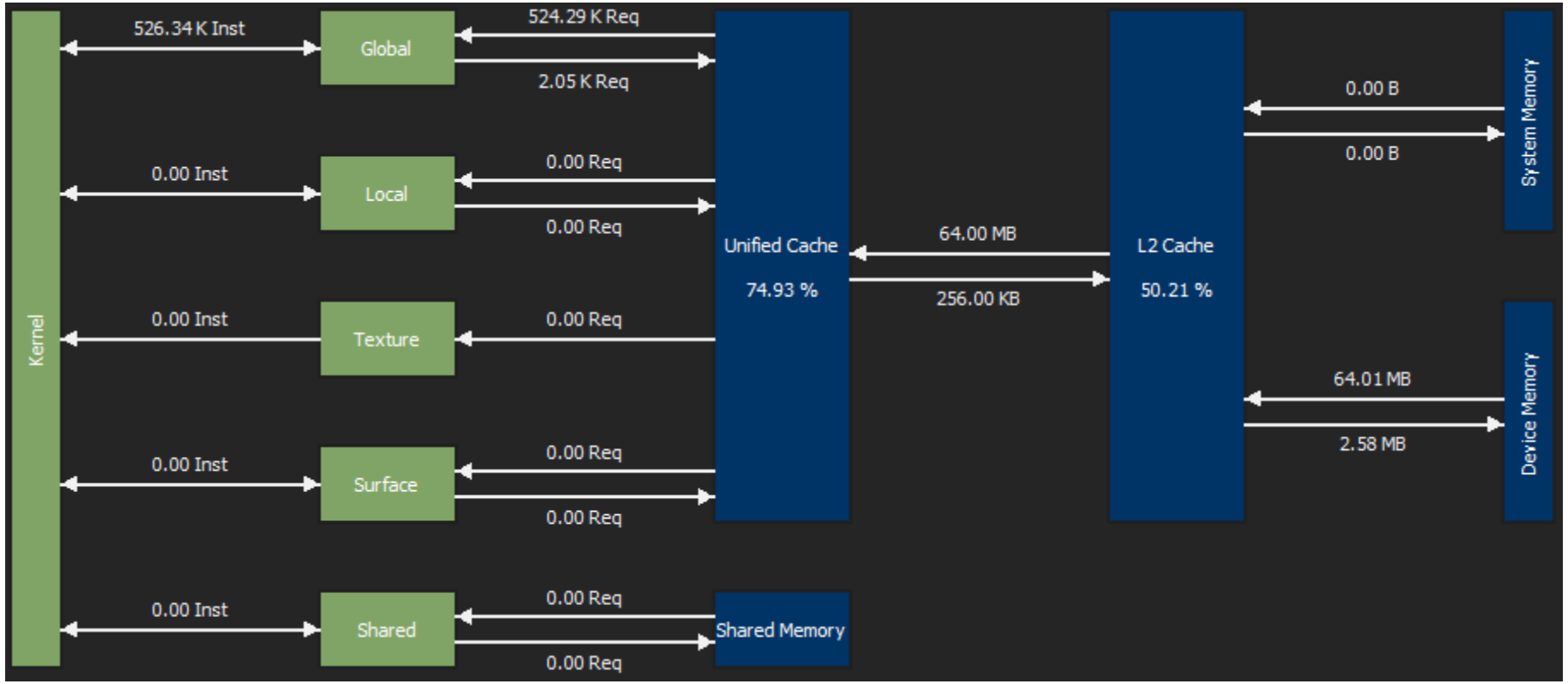


ColumnAfterColumn: Texture vs Global

```
__global__ void device_load_row_after_row(const float4* data, int pitch, int width, int height,
                                          float4* out) {
    float4 sum = make_float4(0,0,0,0);
    int xin = blockIdx.x*blockDim.x + threadIdx.x;
    int yin = blockIdx.y*blockDim.y + threadIdx.y;

    for (int y = yin; y < height; y += blockDim.y) {
        float4 in = data[xin + y*pitch];
        sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
    }
    out[xin + yin*gridDim.x*blockDim.x] = sum;
}
```

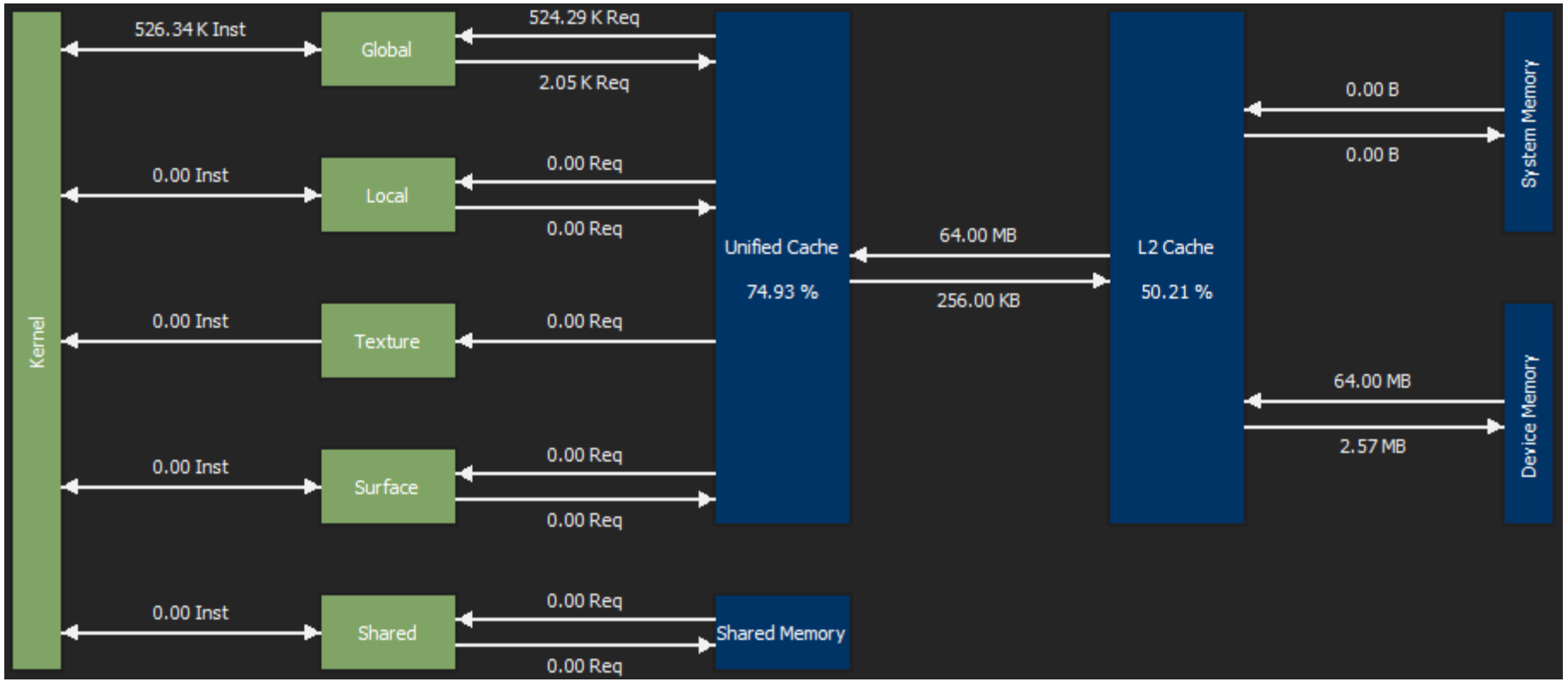
```
float4 in = tex2D(my_tex, xin, y);
```

ColumnAfterColumn: Texture vs Global



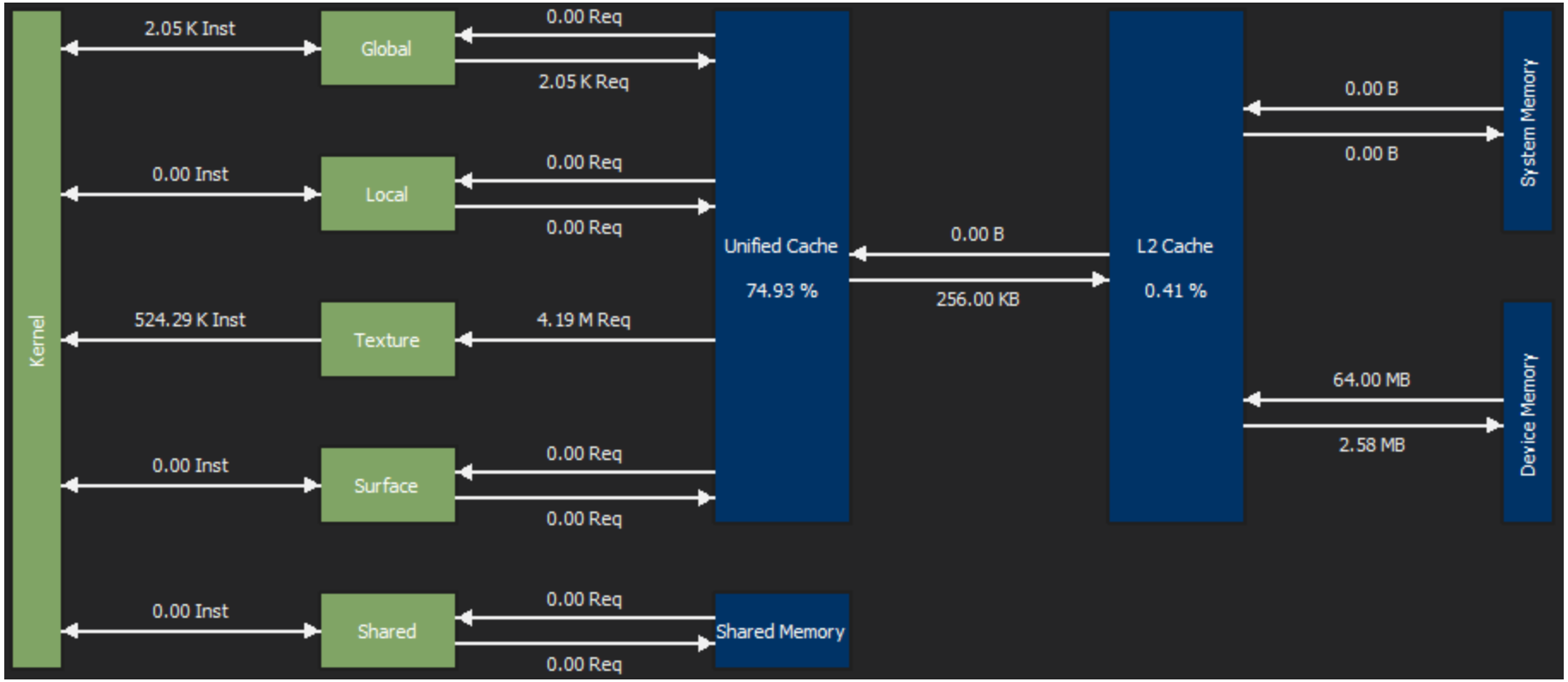
global linear (cudaMalloc) 439GB/s

ColumnAfterColumn: Texture vs Global



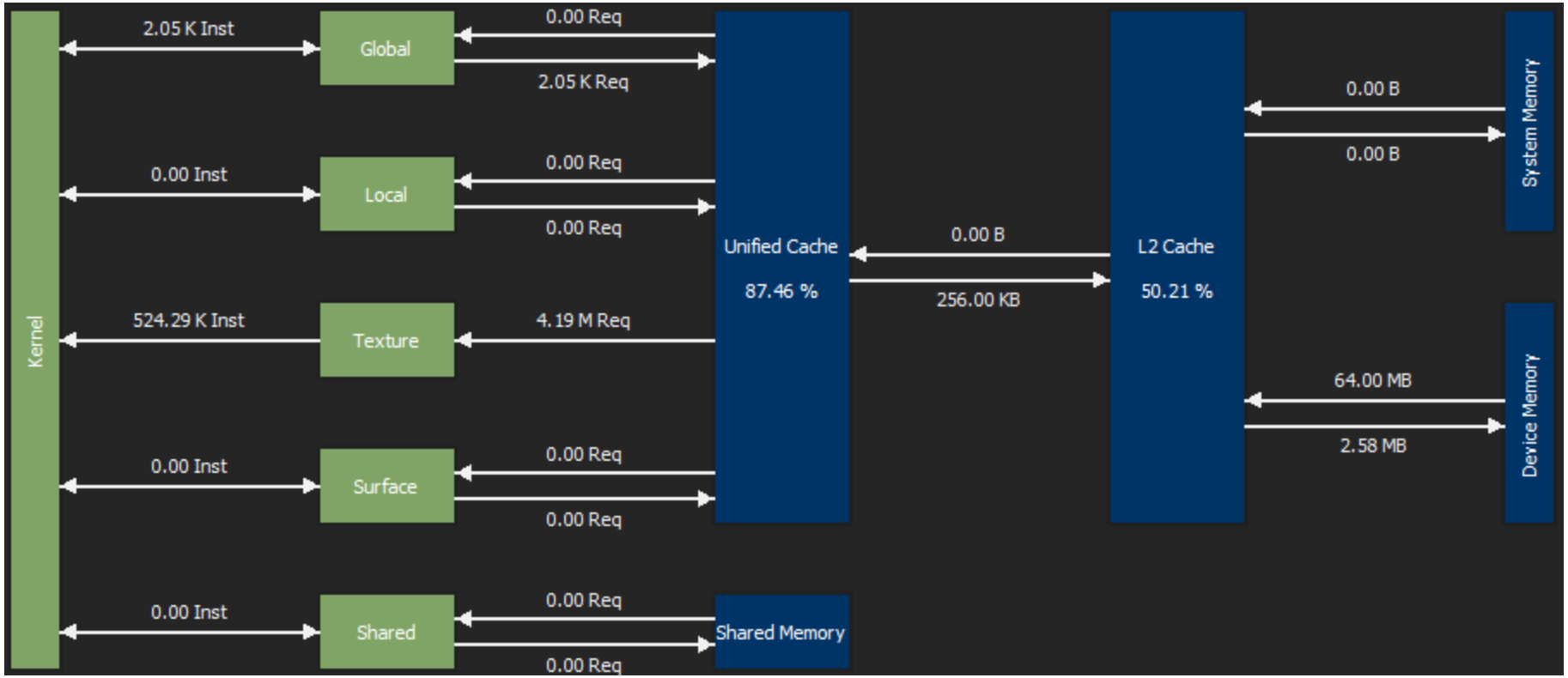
global 2D (cudaMallocPitch) 442GB/s

ColumnAfterColumn: Texture vs Global



texture (cudaArray) 434GB/s

ColumnAfterColumn: Texture vs Global



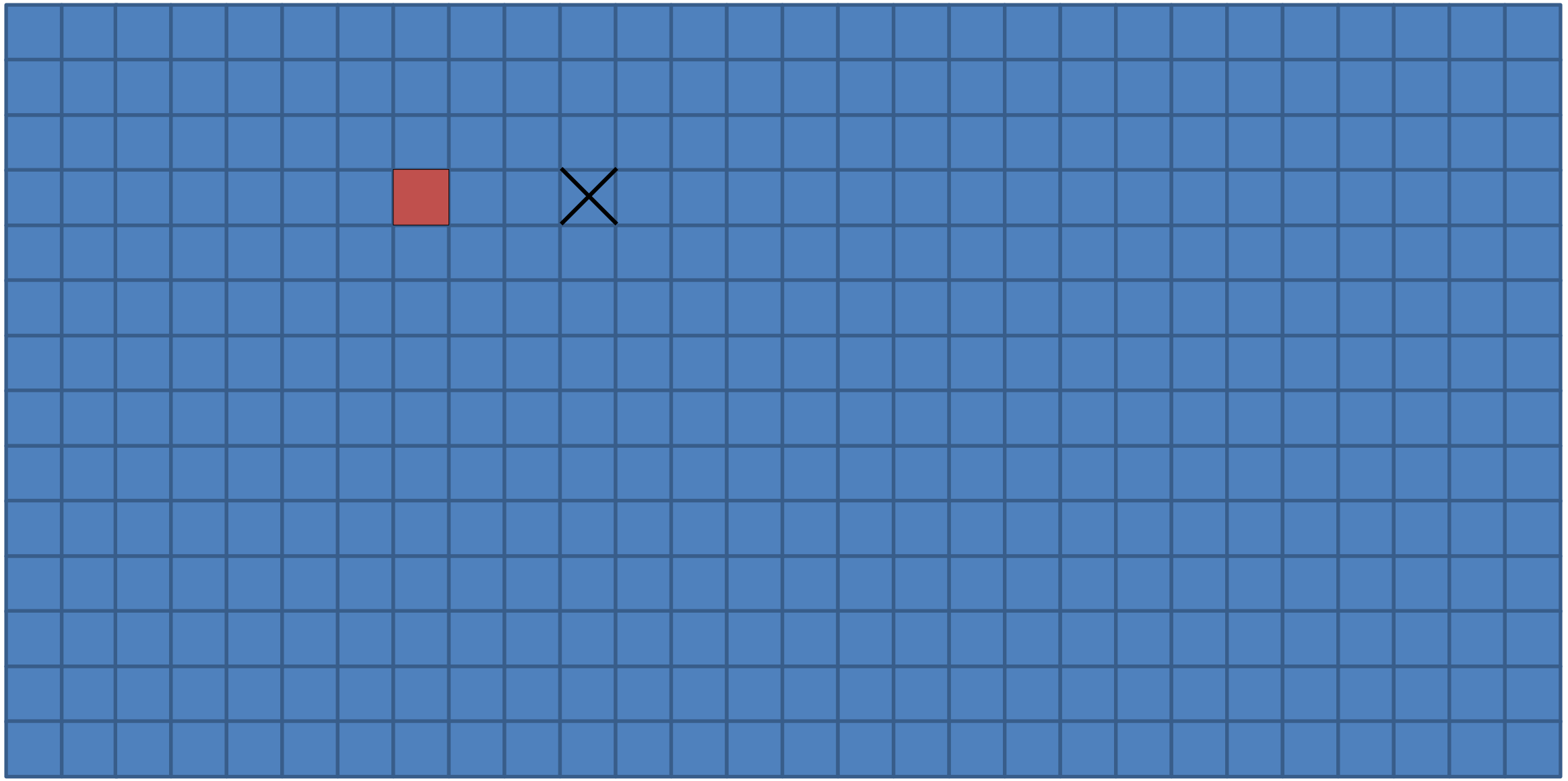
texture (cudaMallocPitch) 428GB/s

Texture vs Global: Sliding

	ms				GB/s		
	Row						
Approach	1080 Ti	Titan V	2080 Ti		1080 Ti	Titan V	2080 Ti
Const Array	0,2540	0,1454	0,1450		264,2	461,5	462,7
Const Array Pitched	0,2540	0,1444	0,1452		264,2	464,8	462,1
Texture	0,2552	0,1546	0,1372		263,0	434,0	489,2
Texture Array	0,2614	0,1587	0,1495		256,7	422,8	448,9
	Column						
Approach	1080 Ti	Titan V	2080 Ti		1080 Ti	Titan V	2080 Ti
Const Array	0,5101	0,1526	0,2080		131,6	439,8	322,6
Const Array Pitched	0,5100	0,1516	0,2123		131,6	442,8	316,1
Texture	0,2530	0,1546	0,1372		265,3	434,1	489,2
Texture Array	0,2805	0,1567	0,1443		239,2	428,3	465,2

- Row access is similarly efficient with all variants
- Column access is divided
  - Consumer cards profit greatly from spatial access pattern in texture
  - Titan V seemingly does not care

Filter X



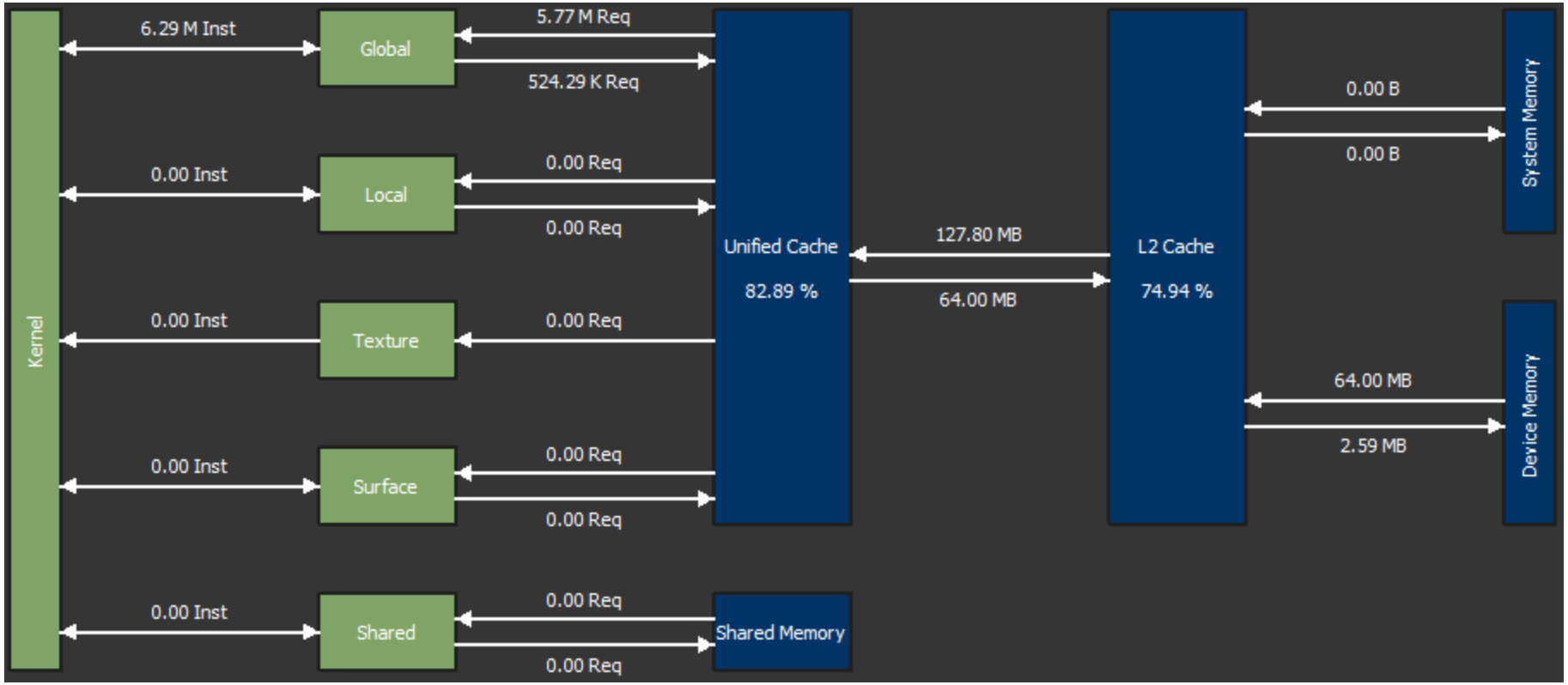
Filter X: Texture vs. Global

```
__global__ void global_filter_x(const uchar4* data, int pitch, int width, int height,
                                uchar4* out, int offset) {
    uchar4 sum = make_uchar4(0, 0, 0, 0);
    int xin = blockIdx.x*blockDim.x + threadIdx.x;
    int yin = blockIdx.y*blockDim.y + threadIdx.y;

    if (xin >= offset && xin < width-offset-1) {
        for (int x = xin-offset; x <= xin+offset; ++x) {
            uchar4 in = data[x + yin*pitch];
            sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
        }
    }
    yin = yin % blockDim.y;
    out[xin + yin*gridDim.x*blockDim.x] = sum;
}
```

No need for if

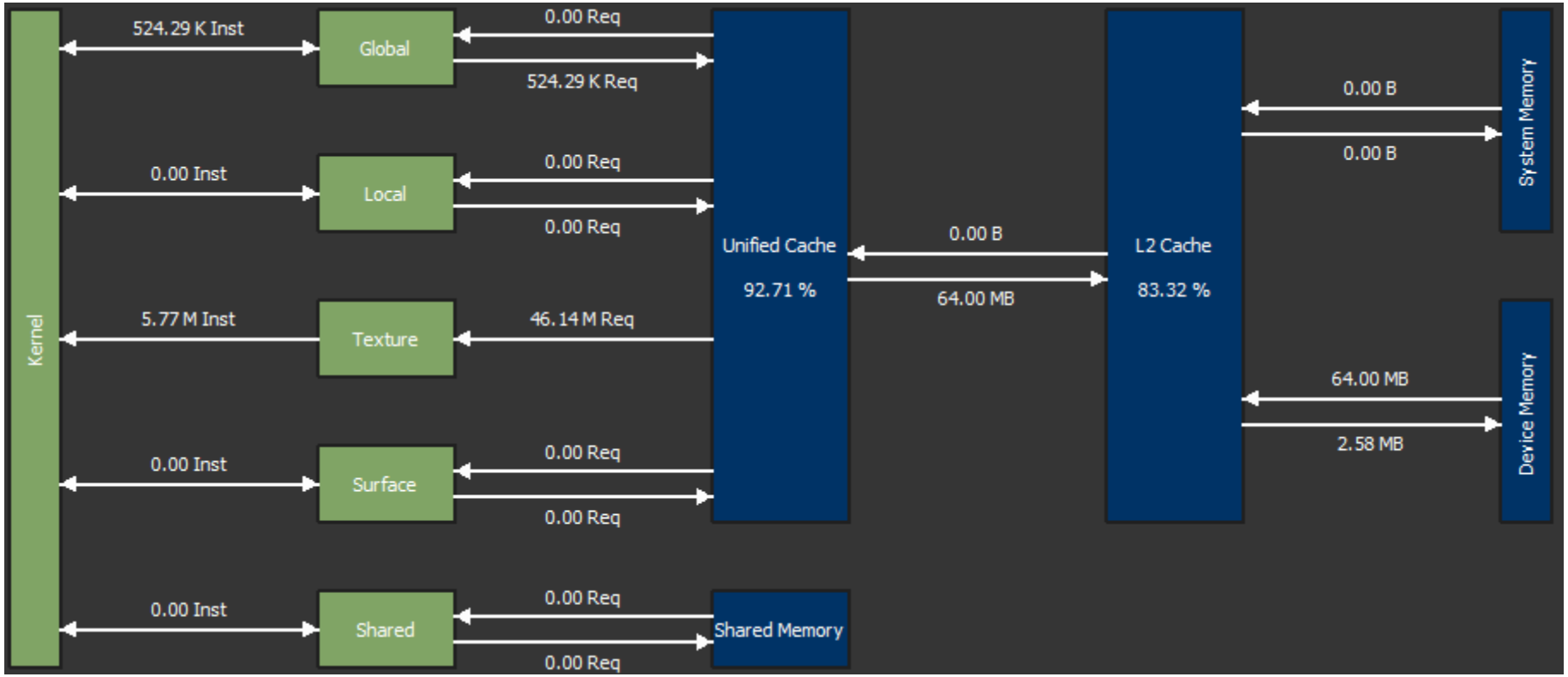
```
uchar4 in = tex2D(my_tex, x, yin);
```

Filter X: Texture vs. Global



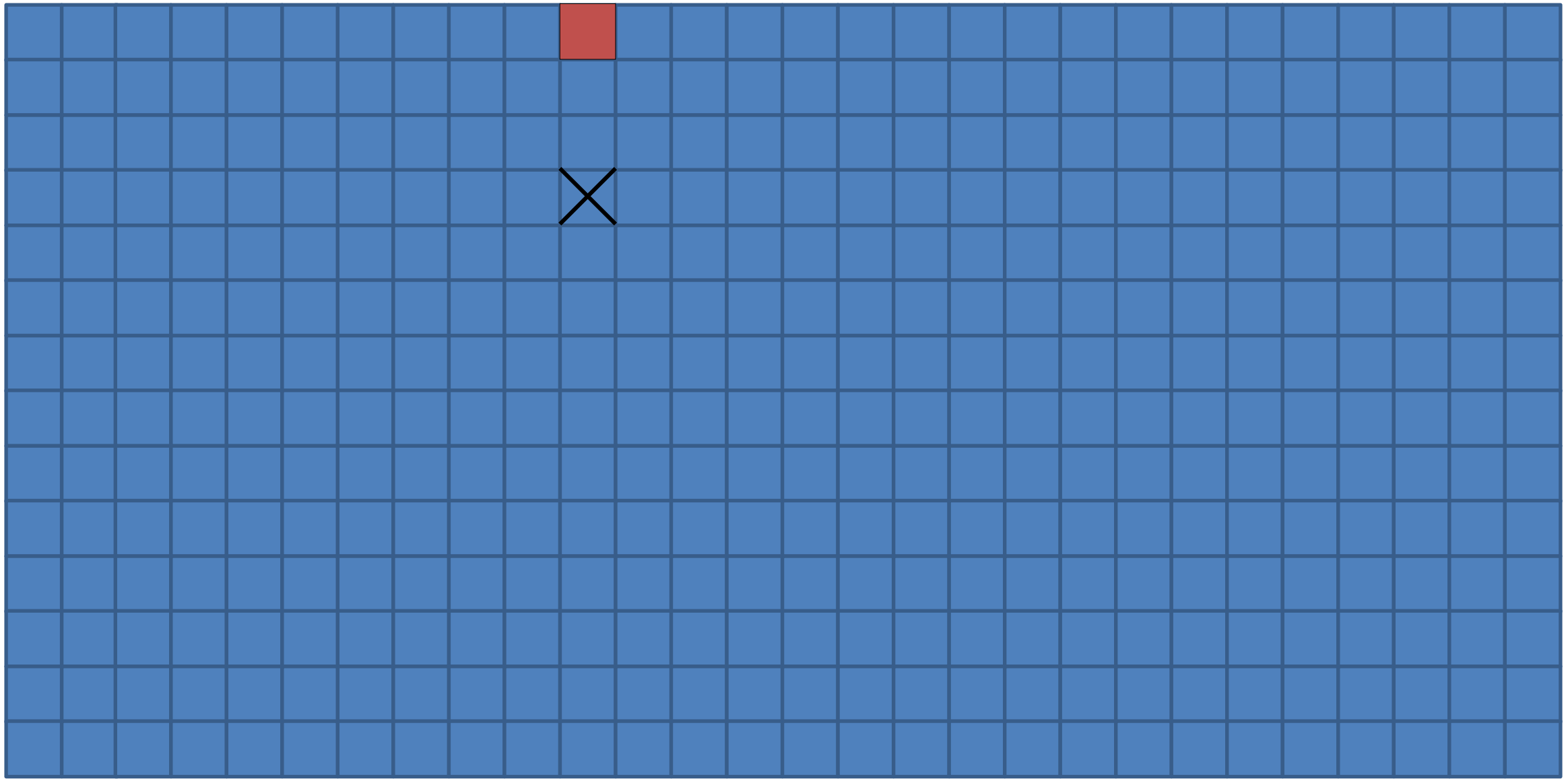
global linear (cudaMalloc) 1602GB/s

Filter X: Texture vs. Global



texture(cudaArray) 1386GB/s

Filter Y



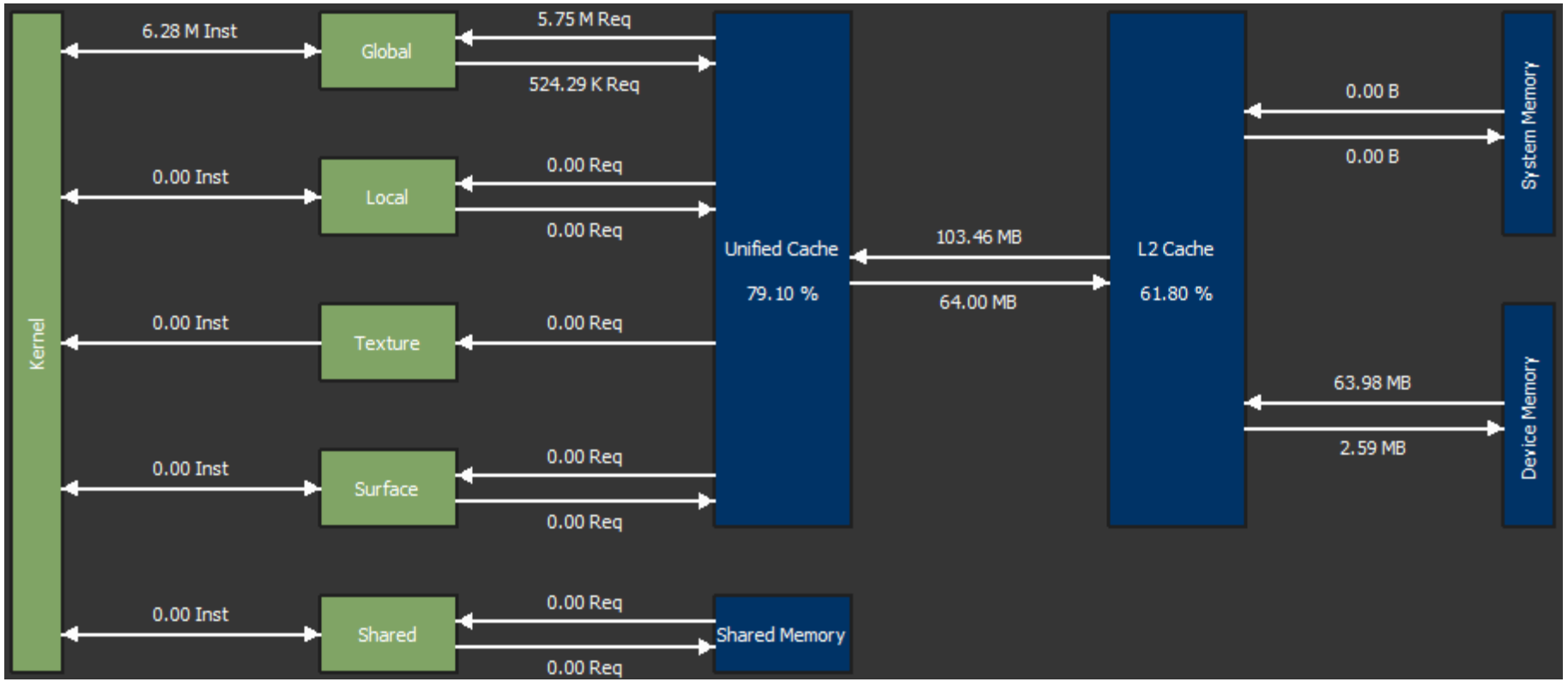
Filter Y: Texture vs. Global

```
__global__ void global_filter_y(const uchar4* data, int pitch, int width, int height,
                                uchar4* out, int offset) {
    uchar4 sum = make_uchar4(0, 0, 0, 0);
    int xin = blockIdx.x*blockDim.x + threadIdx.x;
    int yin = blockIdx.y*blockDim.y + threadIdx.y;

    if (yin >= offset && yin < height-offset-1) {
        for (int y = yin-offset; y <= yin+offset; ++y) {
            uchar4 in = data[xin + y*pitch];
            sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
        }
    }
    yin = yin % blockDim.y;
    out[xin + yin*gridDim.x*blockDim.x] = sum;
}
```

No need for if

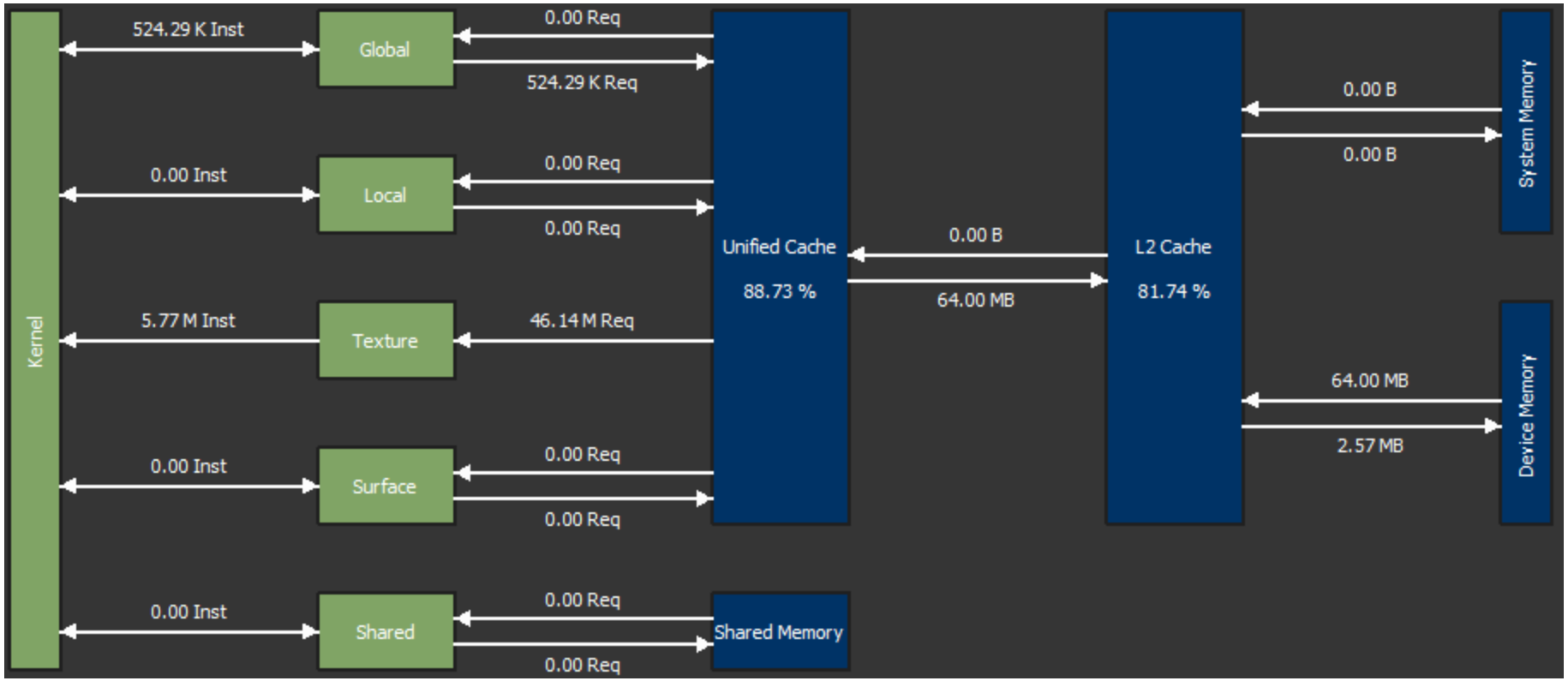
```
uchar4 in = tex2D(my_tex, xin, y);
```

Filter Y: Texture vs. Global



global linear (cudaMalloc) 1560GB/s

Filter Y: Texture vs. Global



texture(cudaArray) 1411GB/s

Texture vs. Global: Filtering

	ms				GB/s		
	X						
Approach	1080 Ti	Titan V	2080 Ti		1080 Ti	Titan V	2080 Ti
Const Array	1,126	0,4608	0,5302		655,4	1602	1392
Const Array Pitched	1,125	0,4588	0,5290		656,4	1609	1237
Texture	0,7414	0,5324	0,6846		955,7	1386	1078
Texture Array	0,7415	0,5304	0,6918		955,5	1392	1067
	Y						
Approach	1080 Ti	Titan V	2080 Ti		1080 Ti	Titan V	2080 Ti
Const Array	0,8999	0,4731	0,5975		820,3	1560	1235
Const Array Pitched	0,8994	0,4731	0,5965		820,8	1560	1237
Texture	0,7229	0,5233	0,6922		1021	1411	1066
Texture Array	0,7332	0,5274	0,7141		1007	1400	1034

- Prior to Volta
  - Texture access always beneficial → linear memory only cached in L2
- Volta +
  - Linear memory outperforms texture memory → linear memory cached in L1 as well

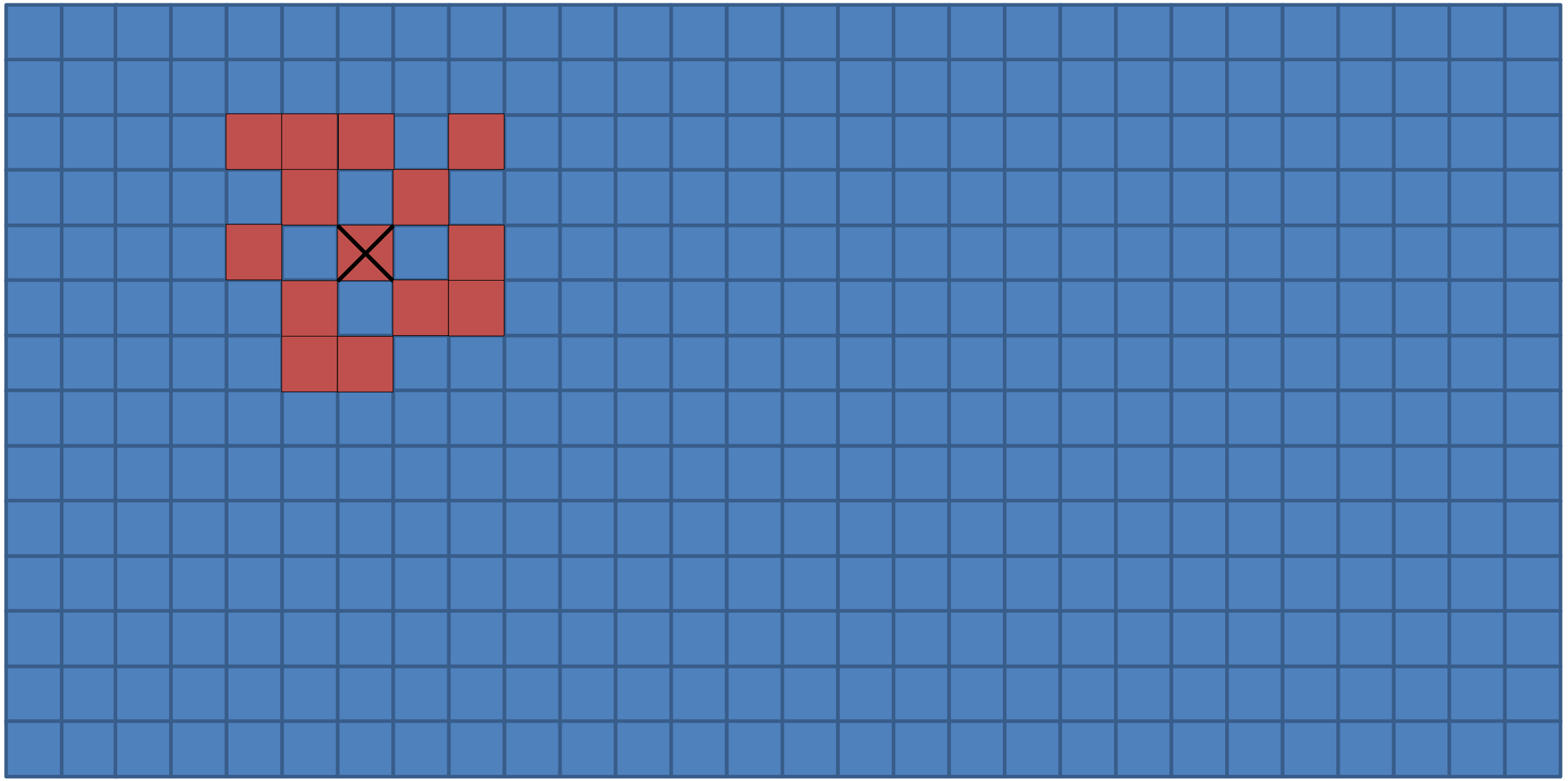
Texture vs. Global: Filtering

- global_lin_filter_x 10.7816ms → 273.872GB/s
  global_2d_filter_x 10.7704ms → 274.157GB/s
  tex_array_filter_x 5.02566ms → **587.542GB/s**
  tex_2d_filter_x 5.73747ms → 514.65GB/s

  global_lin_filter_y 10.1444ms → 291.076GB/s
  global_2d_filter_y 10.1401ms → 291.2GB/s
  tex_array_filter_y 8.09706ms → 364.674GB/s
  tex_2d_filter_y 5.95472ms → **495.874GB/s**

- L2 cache important for all examples
- Texture cache boosts speed
- Filtering along X is faster than Y

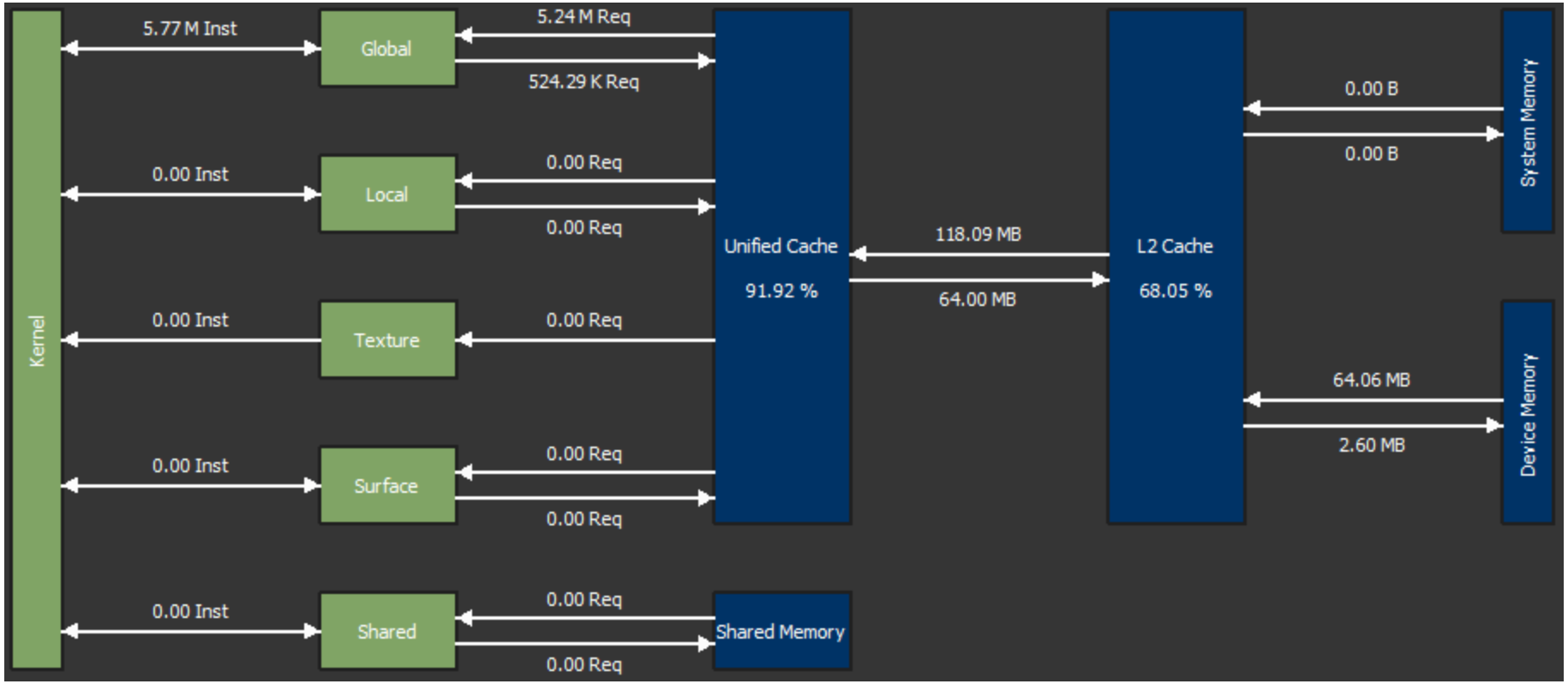
Random Access in Vicinity



Random Sampling: Texture vs. Global

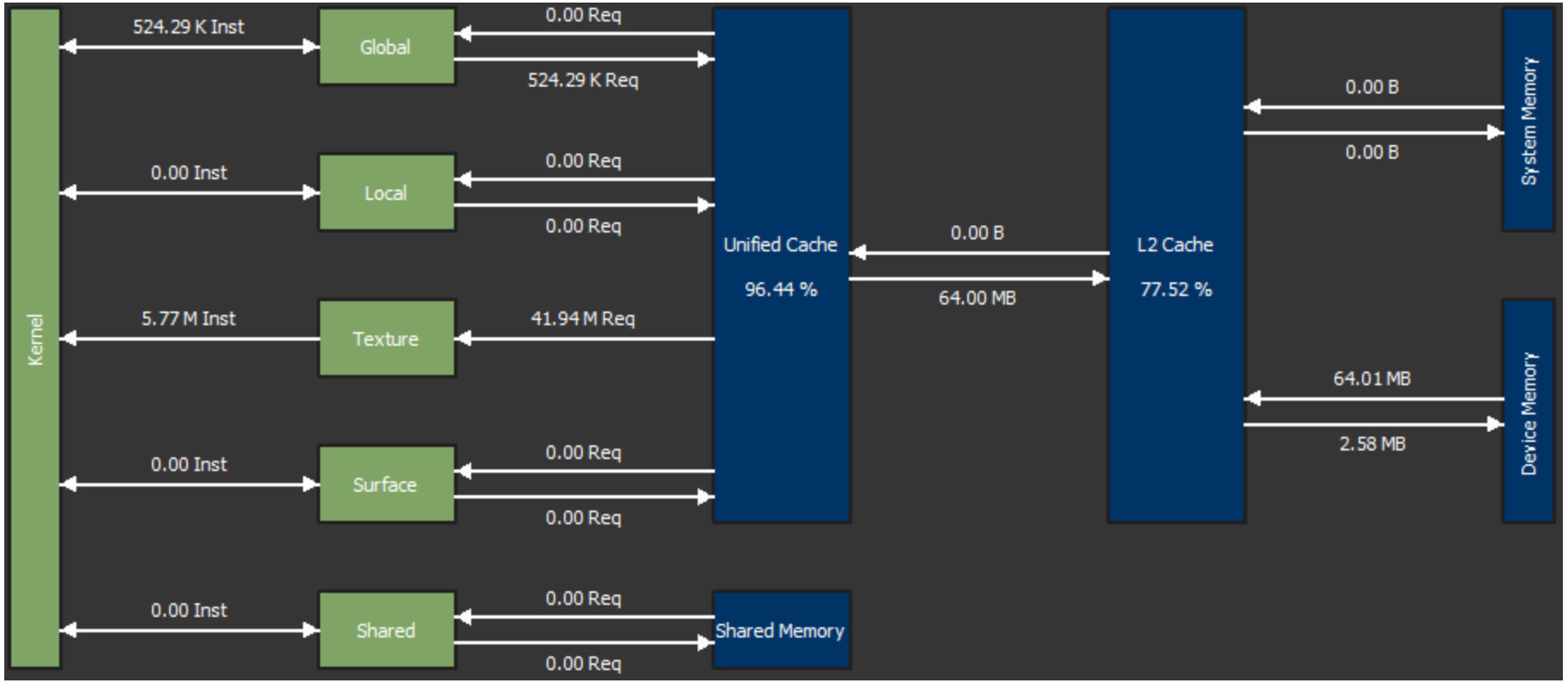
```
template<int Area>
__global__ void device_read_random(const uchar4* data, int pitch, int width, int height,
                                   uchar4* out, int samples) {
    uchar4 sum = make_uchar4(0, 0, 0, 0);
    int xin = blockIdx.x*blockDim.x + Area*(threadIdx.x/Area);
    int yin = blockIdx.y*blockDim.y + Area*(threadIdx.y/Area);

    unsigned int xseed = threadIdx.x*9182 + threadIdx.y*91882  + threadIdx.x*threadIdx.y*811 + 72923181;
    unsigned int yseed = threadIdx.x*981  + threadIdx.y*124523 + threadIdx.x*threadIdx.y*327 + 98721121;

    for (int sample = 0; sample < samples; ++sample) {
        unsigned int x = xseed%Area;
        unsigned int y = yseed%Area;
        xseed = (xseed * 1587);
        yseed = (yseed * 6971);
        uchar4 in = data[xin + x + (yin + y)*pitch];
        sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
    }

    yin = (yin + threadIdx.y%Area) % blockDim.y;
    out[xin + threadIdx.x%Area + yin*width] = sum;
}
```

Random Sampling: Texture vs. Global



global linear(cudaMalloc) 1014GB/s

Random Sampling: Texture vs. Global



texture(cudaArray) 1170GB/s

Random Sampling: Texture vs. Global

	ms				GB/s		
Approach	1080 Ti	Titan V	2080 Ti		1080 Ti	Titan V	2080 Ti
Const Array	2,3940	0,6615	0,7749		280,4	1014	866,1
Const Array Pitched	2,3920	0,6615	0,7740		280,5	1014	867,0
Texture	1,4840	0,5734	0,7436		452,3	1170	902,5
Texture Array	1,4830	0,5683	0,7492		452,6	1181	895,8

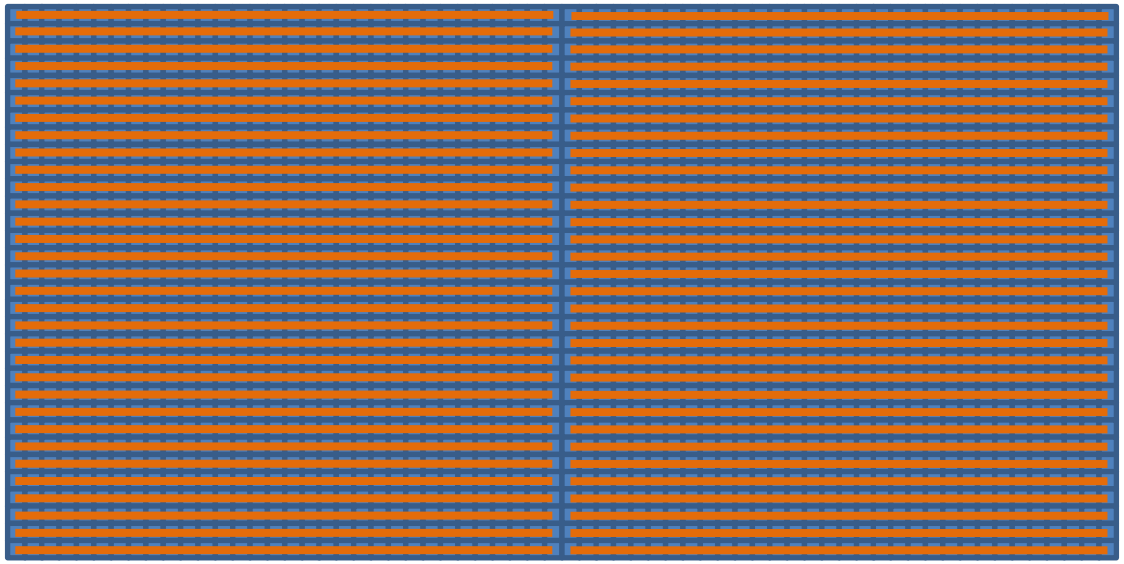
- L2 cache important for all examples
- Texture cache boosts speed

Texture vs. Global Conclusion

- No so clear anymore
- Before $CC < 7.x$
  - Texture performs as good and sometimes better
  - Put less stress on L2 cache
  - L1 free for other tasks
  - Features: border handling, interpolation, conversion
- Now $CC \geq 7.0$
  - Unified cache (L1 + Tex)
  - Much more advanced caching
    - Utilizes unified cache very efficiently
  - Textures still perform best for spatial access pattern or where linear access clearly fails
  - But can also be slower if access pattern and cache hits favor linear memory

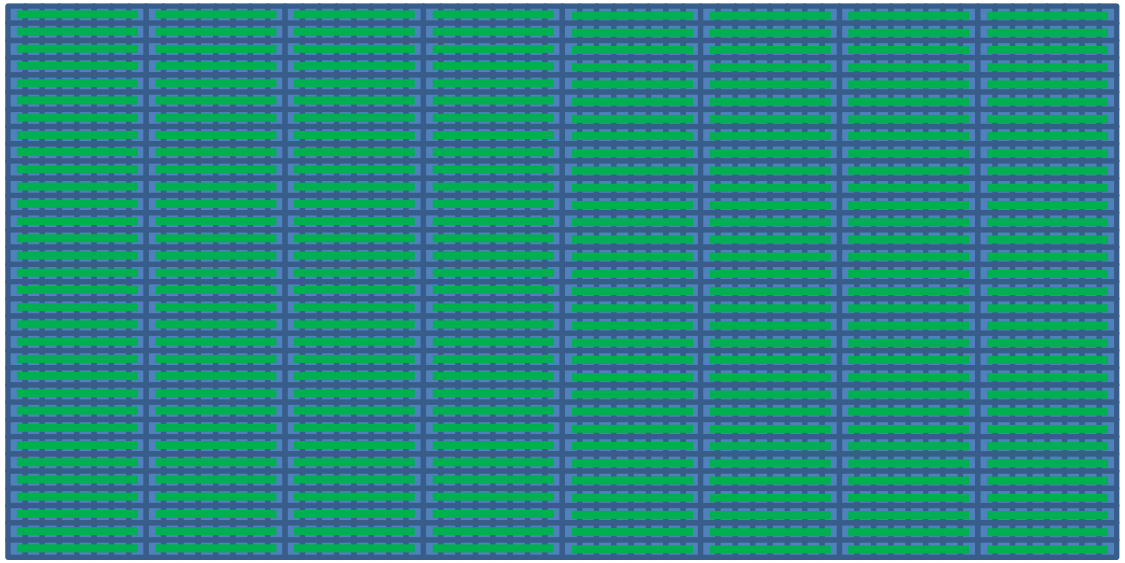
Array vs. Pitched Textures

- Pitched global access: cache line size 128 byte



Array vs. Pitched Textures

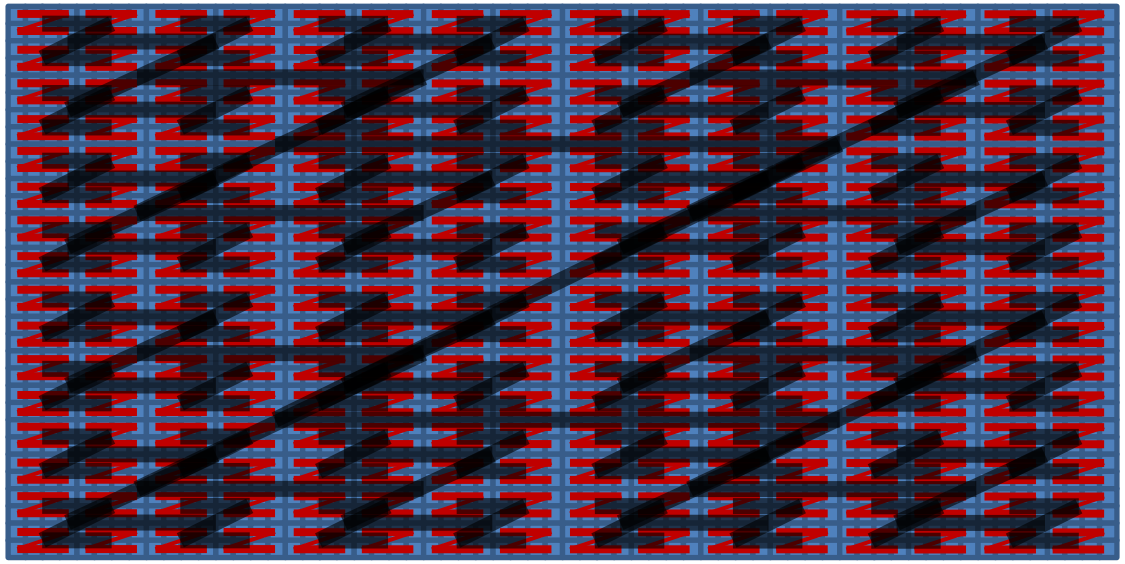
- Pitched global access: cache line size 128 byte
- Pitched texture access: cache line size 32 byte



Array vs. Pitched Textures

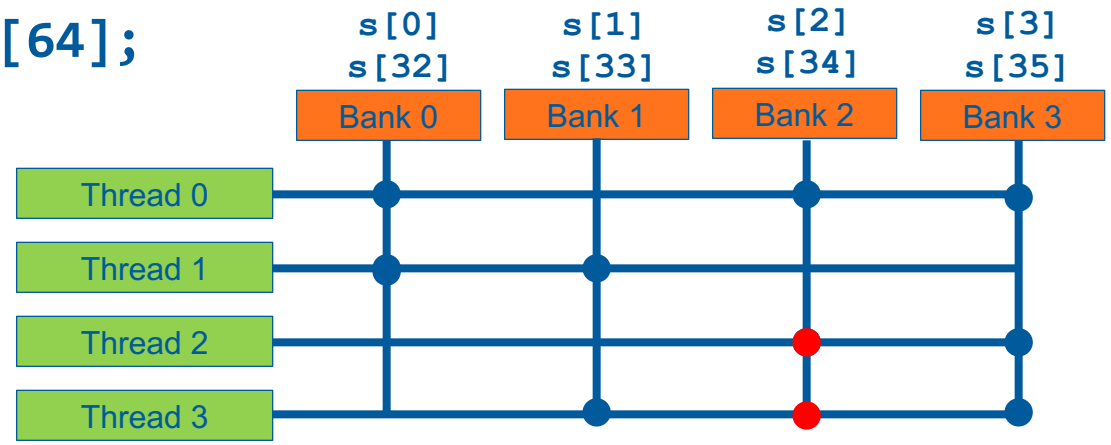
- Pitched global access: cache line size 128 byte
- Pitched texture access: cache line size 32 byte
- Array texture access: space filling curve (maybe Z curve)

  + cache line size 32 byte



Shared Memory - Recap

- Shared access within one block (lifetime: block)
- Located on multiprocessor → very fast
- Limited available size on multiprocessor
- Crossbar: simultaneous access to distinct banks
- `__shared__ int s[64];`



Shared Memory

- Accessed via crossbar → access pattern important
- Different behavior for different architectures
  - Fermi $CC = 2.x$

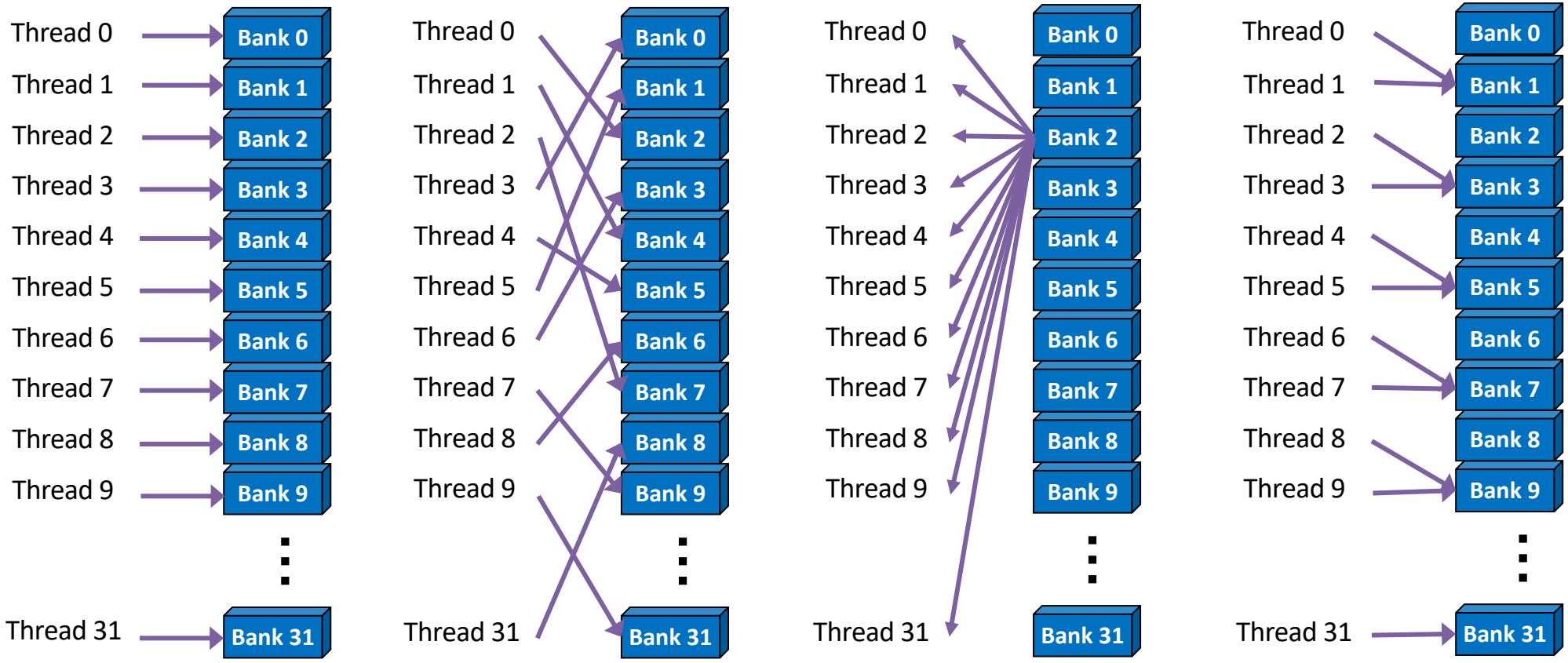
    bank = (address/4) % 32
    - Access issued per 32 threads
    - 32 banks a 32 bits per 2 clock cycles
  - Kepler $CC = 3.x$

    bank = (address/8) % 32
    - Access issued per 32 threads
    - 32 banks a 64 bits per clock cycle
      - 64-bit mode: two threads can access any part of a 64-bit word
      - 32-bit mode: two threads can access any part of a 64-bit word which would fall in the same bank
      - Mode can be set using the CUDA API
  - Maxwell and later $CC \geq 5.0$

    bank = (address/4) % 32
    - Access issued per 32 threads
    - 32 banks a 32 bits per clock cycle

- Key to good performance is the access pattern
  - Conflict free access
    - All threads within a warp access different banks (mind Kepler exceptions)
  - Multicast
    - All threads accessing the same 32-/64-bit word are served with one transaction
  - Serialization
    - Multiple access conflicts will be serialized
- Introduce padding to avoid bank conflicts

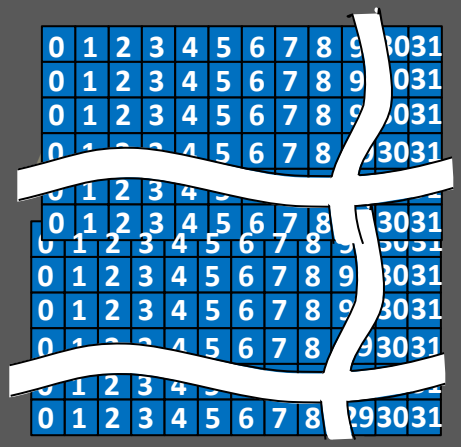
Shared Memory Access Patterns



conflict free access | conflict free access | broadcast | two-way bank conflict (different words)

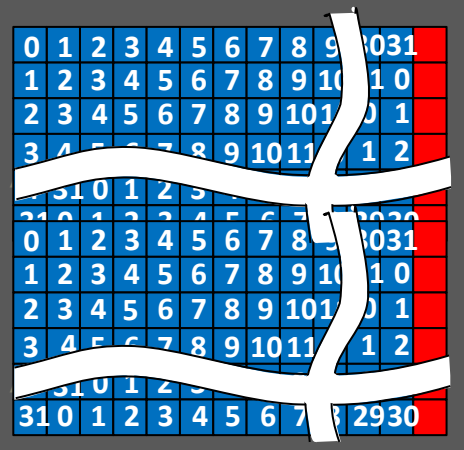
Shared Memory Access Patterns

```
__global__ void kernel(...) {
    __shared__ float mydata[32*32];
    ...
    float sum = 0;
    for (uint i = 0; i < 32; ++i)
        sum += mydata[threadIdx.x + i*32];
    ...
    sum = 0;
    for (uint i = 0; i < 32; ++i)
        sum += mydata[threadIdx.x*32 + i];
    ...
}
```



Shared Memory Access Patterns

```
__global__ void kernel(...) {
    __shared__ float mydata[32*(32 + 1)];
    ...
    float sum = 0;
    for (uint i = 0; i < 32; ++i)
        sum += mydata[threadIdx.x + i*33];
    ...
    sum = 0;
    for (uint i = 0; i < 32; ++i)
        sum += mydata[threadIdx.x*33 + i];
    ...
}
```



Shared Memory Use Cases

- Inter-thread communication

```
__global__ void kernel(...) {
    __shared__ bool run;
    run = true;
    while (run) {
        __syncthreads();
        if (found_it())
            run = false;
        __syncthreads();
    }
}
```

Shared Memory Use Cases

- Inter-thread communication
- Reduce global memory access → manual cache

```
__global__ void kernel(float* global_data, ...) {
    extern __shared__ float data[];
    uint linid = blockIdx.x*blockDim.x + threadIdx.x;
    // load
    data[threadIdx.x] = global_data[linid];
    __syncthreads();
    for (uint it = 0; it < max_it; ++it)
        calc_iteration(data); // calc
    __syncthreads();
    // write back
    global_data[linid] = data[threadIdx.x];
}
```

Shared Memory Use Cases

- Inter-thread communication
- Reduce global memory access → manual cache
- Adjust global memory access pattern

```
__global__ void transpose(float* global_data, float* global_data2) {
    extern __shared__ float data[];
    uint linid1 = blockIdx.x*32 + threadIdx.x + blockIdx.y*32*width;
    uint linid2 = blockIdx.x*32*width + threadIdx.x + blockIdx.y*32;
    for (uint i = 0; i < 32; ++i)
        data[threadIdx.x + i*33] = global_data[linid1 + i*width];
    __syncthreads();
    for (uint j = 0; j < 32; ++j)
        global_data2[linid2 + j*width] = data[threadIdx.x*33 + j] ;
}
```

Shared Memory Use Cases

- Inter-thread communication
- Reduce global memory access → manual cache
- Adjust global memory access pattern
- Index access

```
__global__ void kernel(...) {
    __shared__ uint allmydata[8*BlockSize];
    uint *mydata = allmydata + 8*threadIdx.x;
    for (uint i = 0; i < 8; ++i)
        mydata[i] = complexfunc(i, threadIdx.x);
    uint res = 0;
    for (uint i = 0; i < 64; ++i)
        res += secondfunc(mydata[(threadIdx.x + i) % 8],
                          mydata[i*threadIdx.x % 8]);
}
```

Shared Memory Use Cases

- Inter-thread communication
- Reduce global memory access → manual cache
- Adjust global memory access pattern
- Index access
- Combine costly operations

```
__global__ void kernel(uint *global_count, ...) {
    __shared__ uint blockcount;
    blockcount = 0;
    __syncthreads();
    uint myoffset = atomicAdd(&blockcount, myadd);
    __syncthreads();
    if (threadIdx.x == 0)
        blockcount = atomicAdd(global_count, blockcount);
    __syncthreads();
    myoffset += blockcount;
}
```

Registers vs. Shared Memory vs. Global Memory