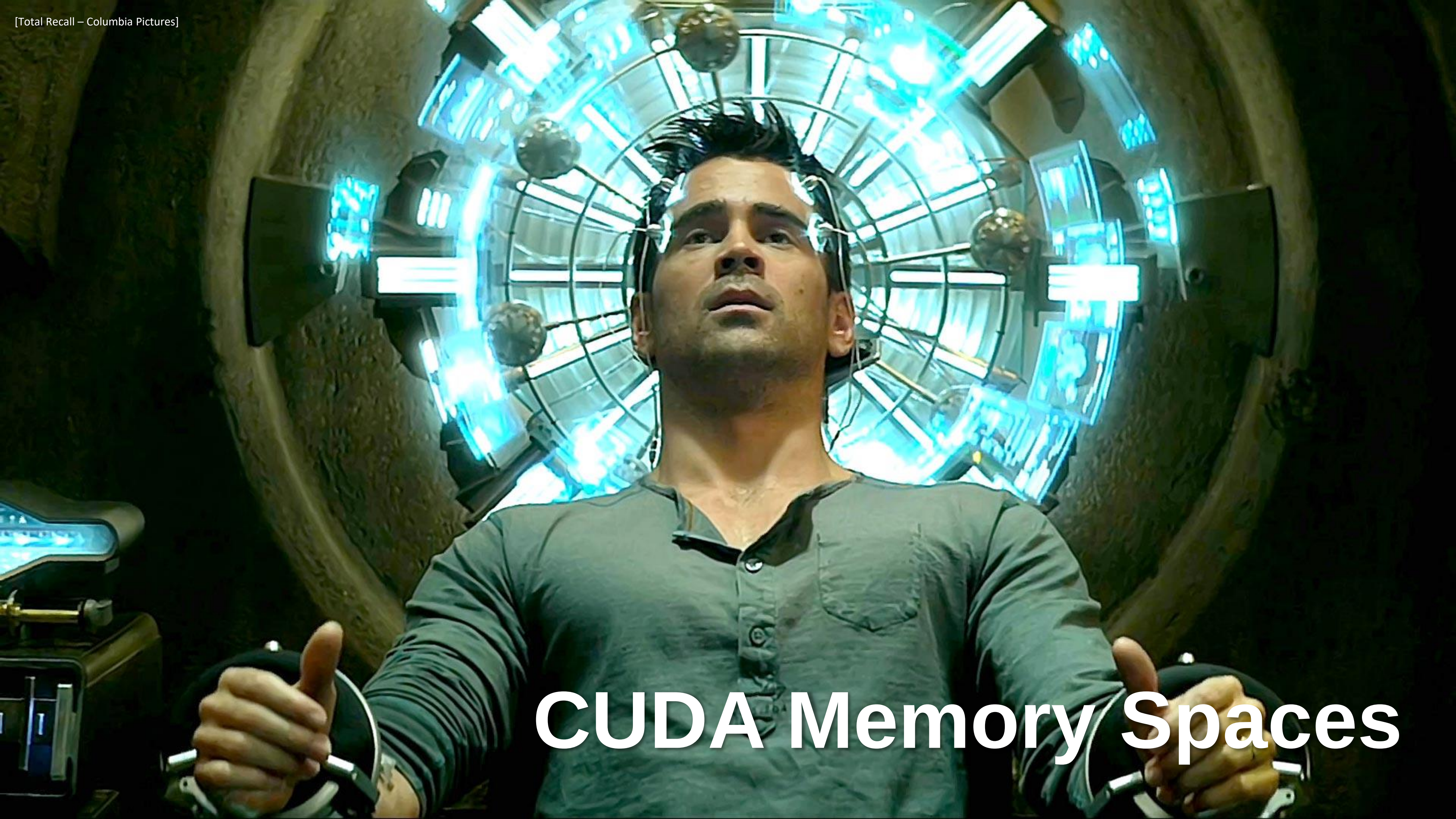
[Total Recall – Columbia Pictures]



CUDA Memory Spaces

CUDA GPU with Caches

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: ~ 180-280 GB/s
  - Non-cached coalesced access: 300 cycles/ns
  - L2 cached access: 160 cycles/ns
  - L1 caches access: 20 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

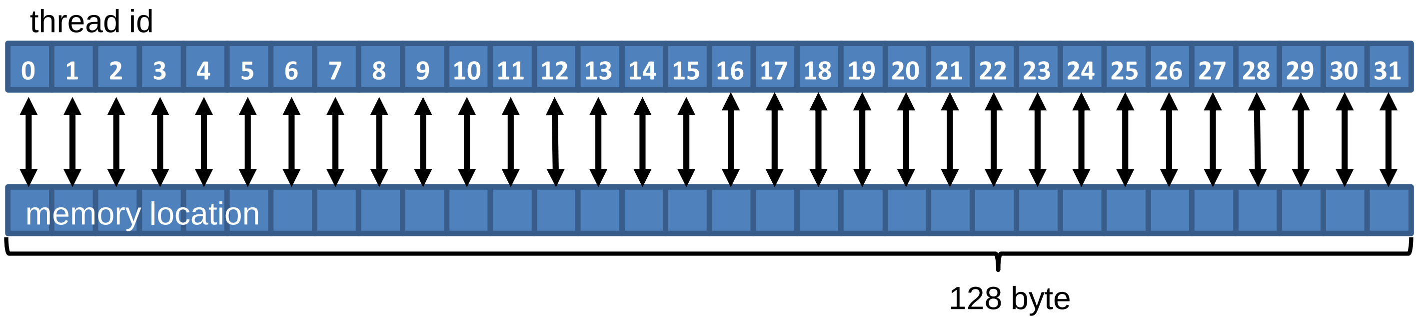
Global memory operations

- Cacheline Size
  - L1: 128 Byte
  - L2: 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

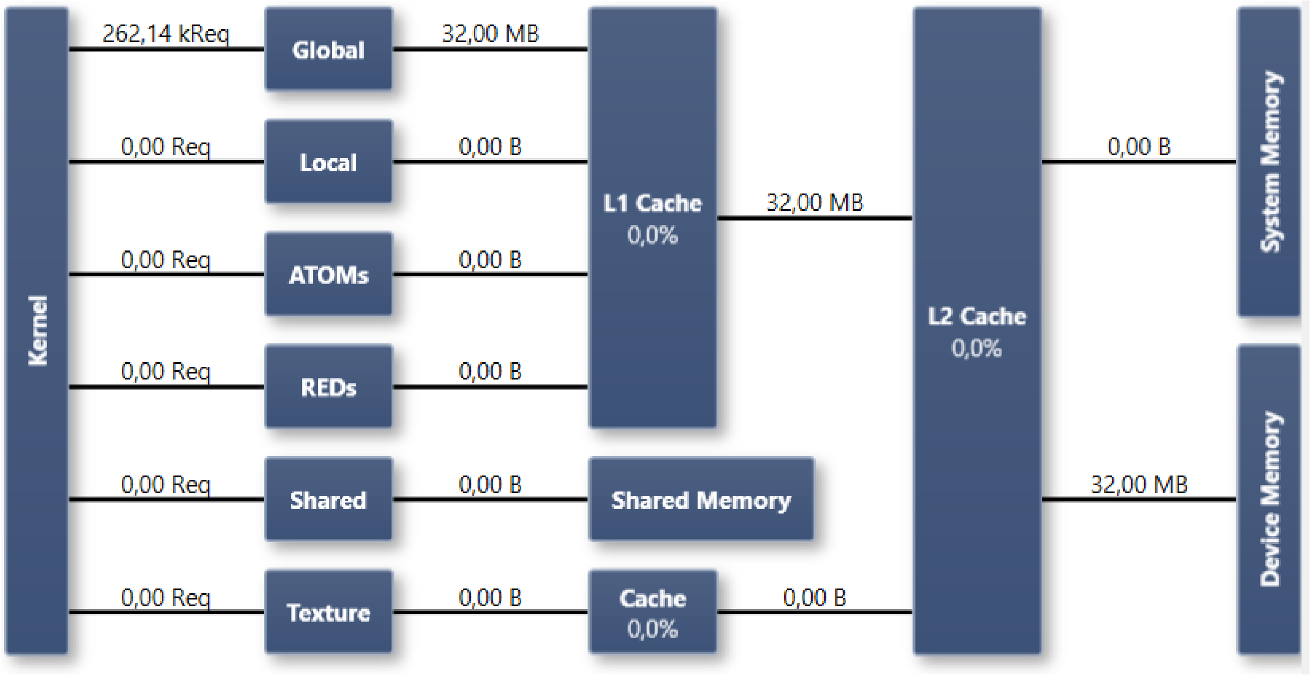
Granularity Example 1/4

```
__global__ void testCoalesced(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];
}
```

testCoalesced done in 0.240736ms -> 69.6914GB/s



Granularity Example 1/4

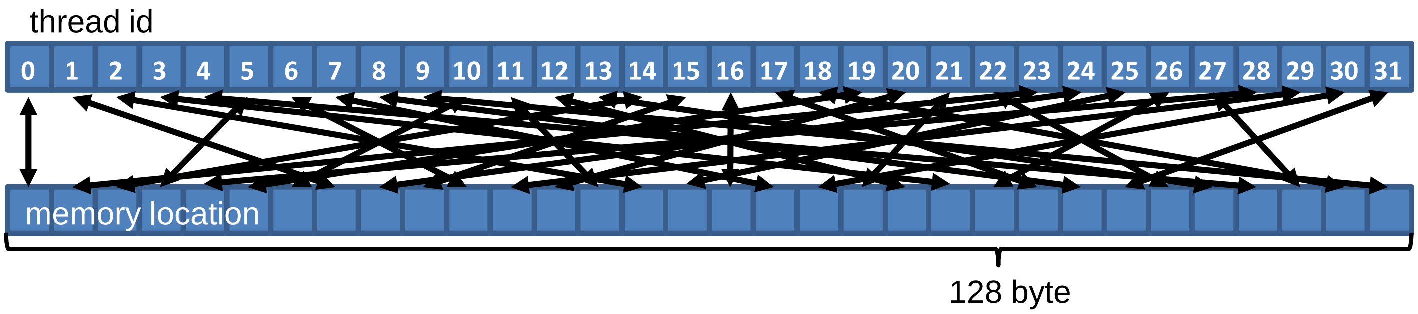


Coalesced

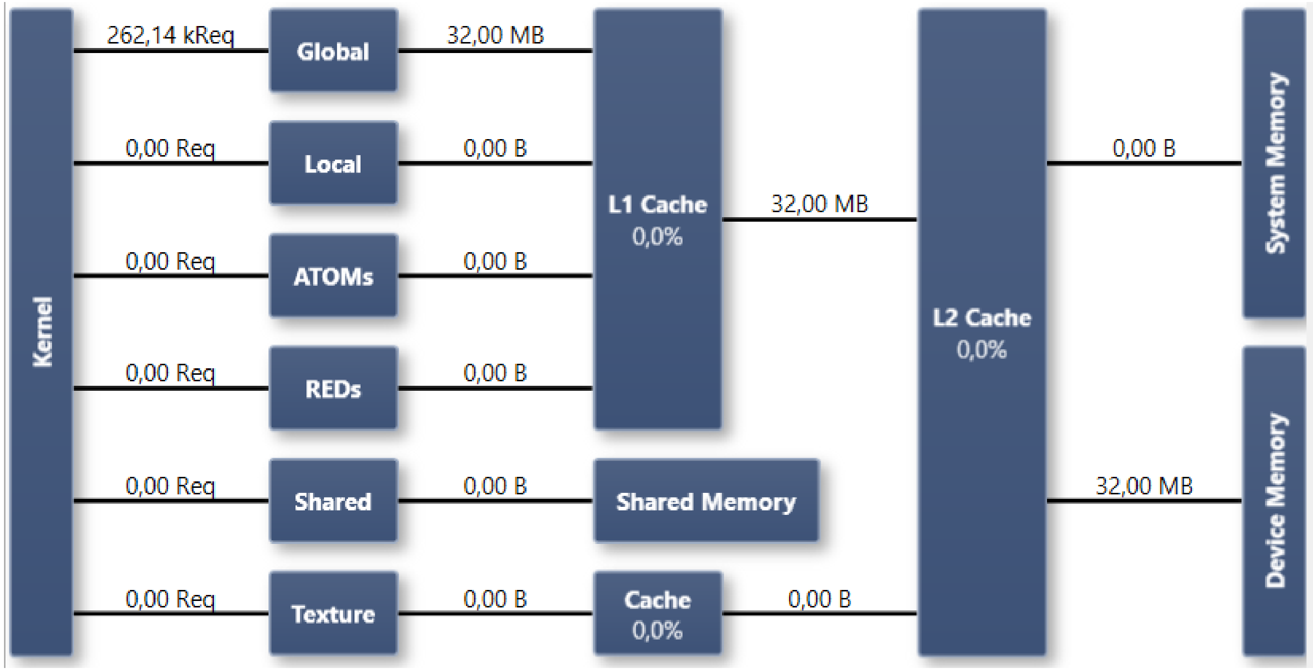
Granularity Example 2/4

```
__global__ void testMixed(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];
}
```

```
testCoalesced done in 0.240736ms -> 69.6914GB/s
testMixed done in 0.24736ms -> 67.8251GB/s
```



Granularity Example 2/4

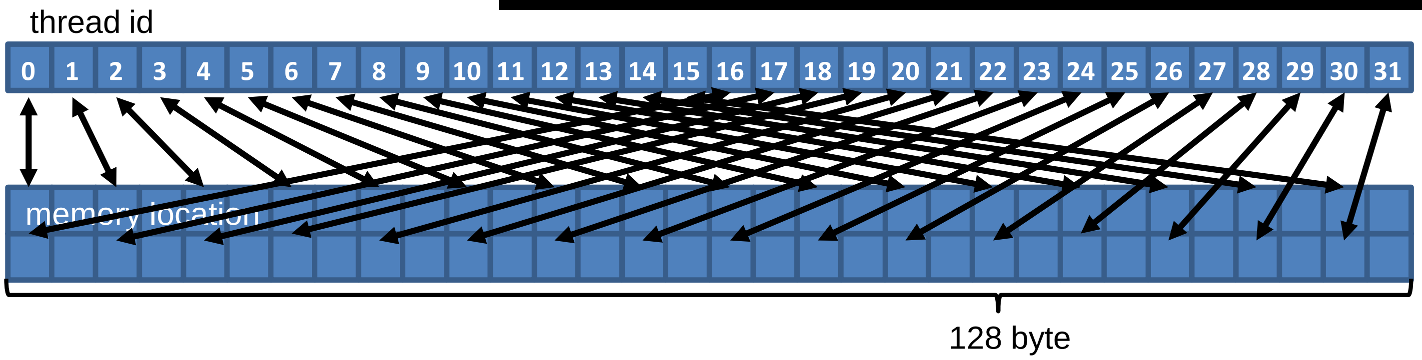


Mixed

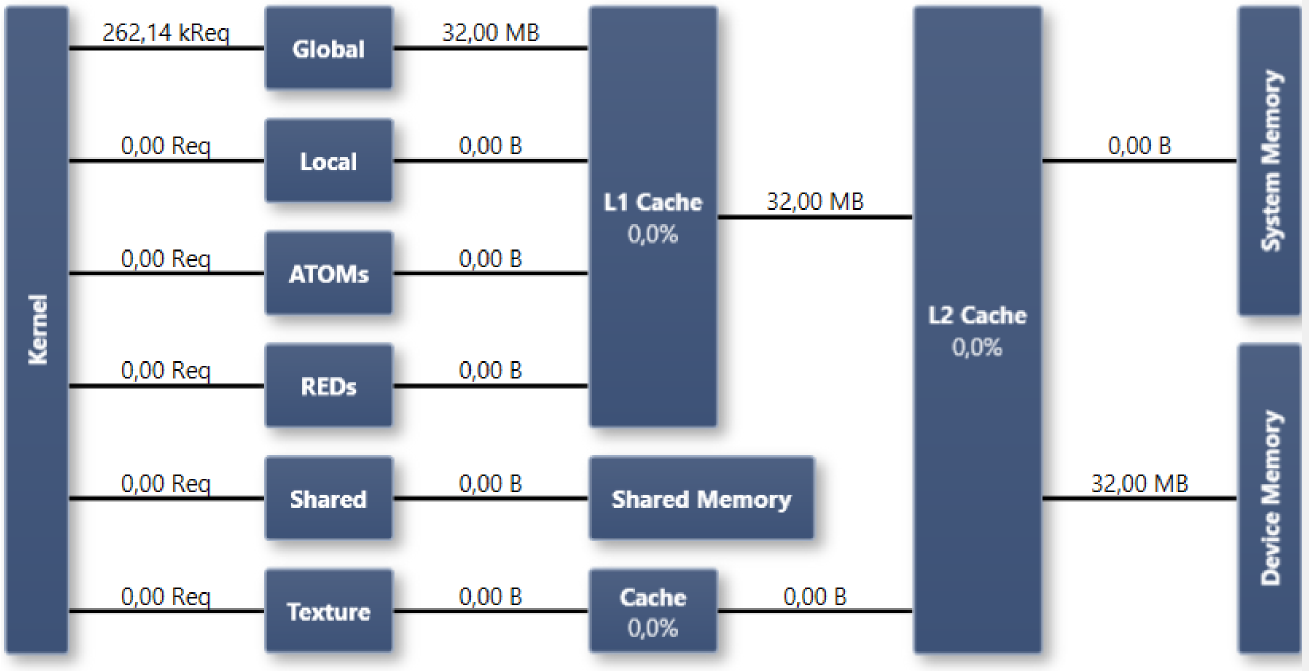
Granularity Example 3/4

```
template<int offset>
__global__ void testOffset(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];
}
```

testCoalesced -> 69.69GB/s
testOffset<2> -> 40.80GB/s testOffset<4> -> 20.27GB/s
testOffset<8> -> 10.06GB/s testOffset<32> -> 6.54GB/s

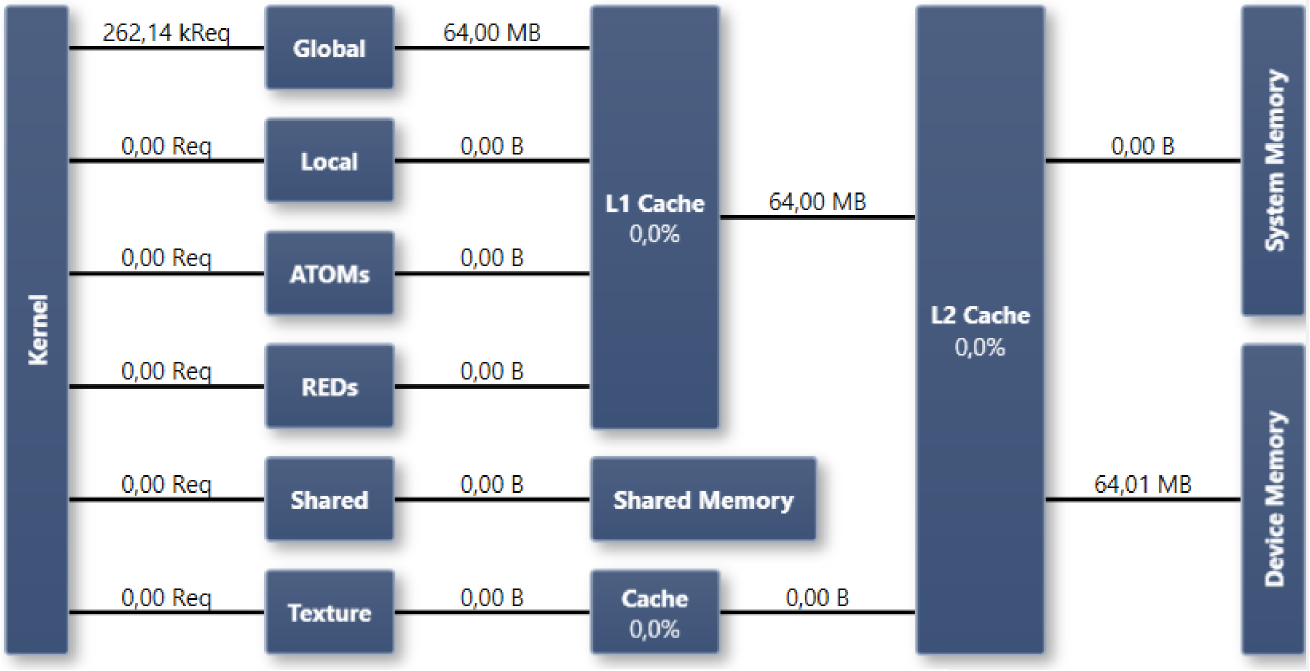


Granularity Example 3/4



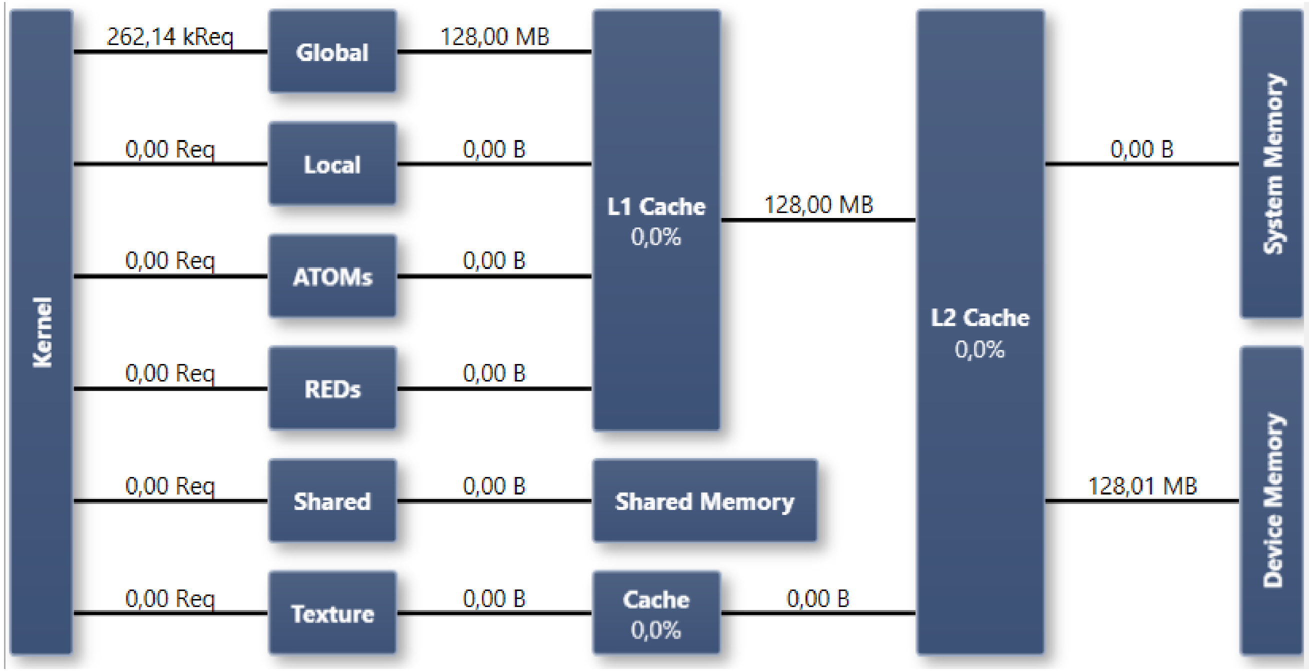
Coalesced 69.69GB/s

Granularity Example 3/4



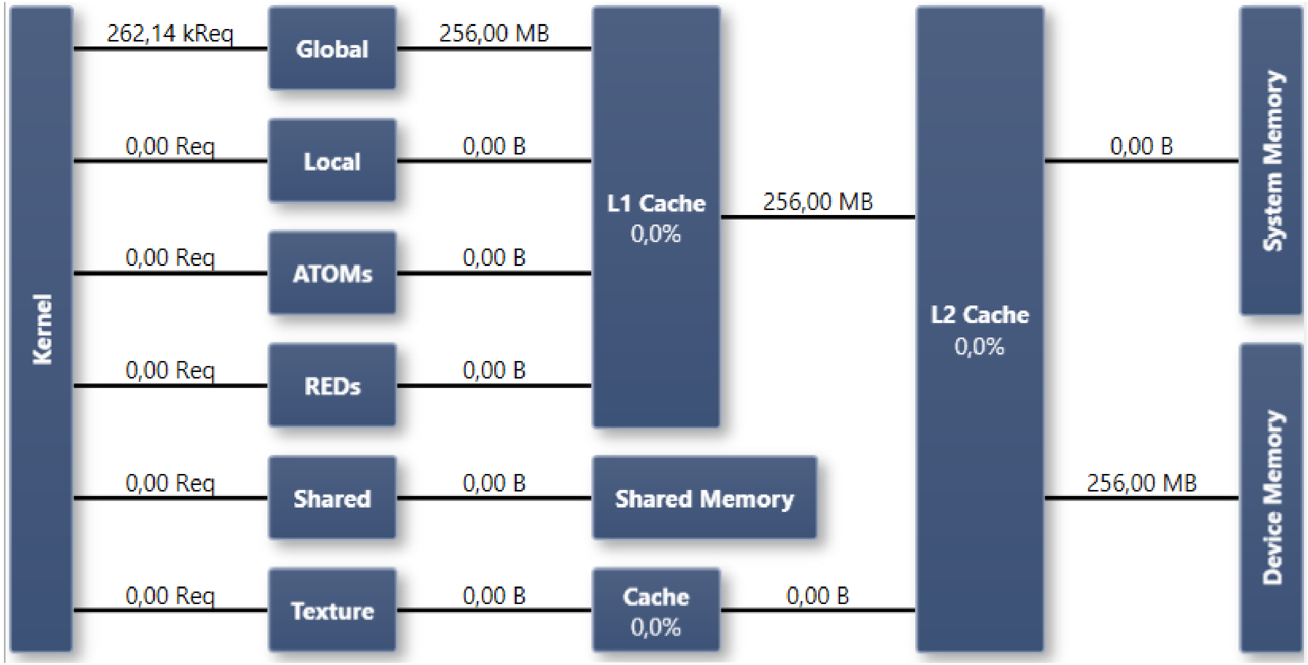
Offset 2 40.80GB/s (0.58)

Granularity Example 3/4



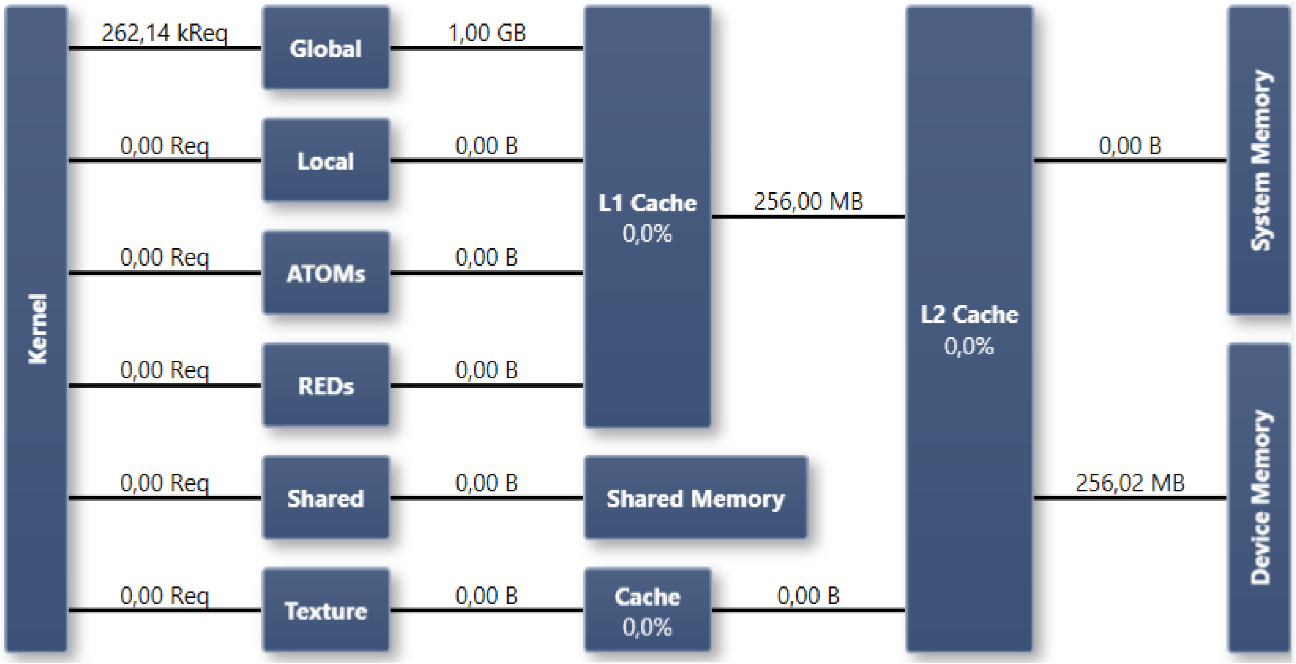
Offset 4 20.27GB/s (0.29)

Granularity Example 3/4



Offset 8 10.06GB/s (0.14)

Granularity Example 3/4



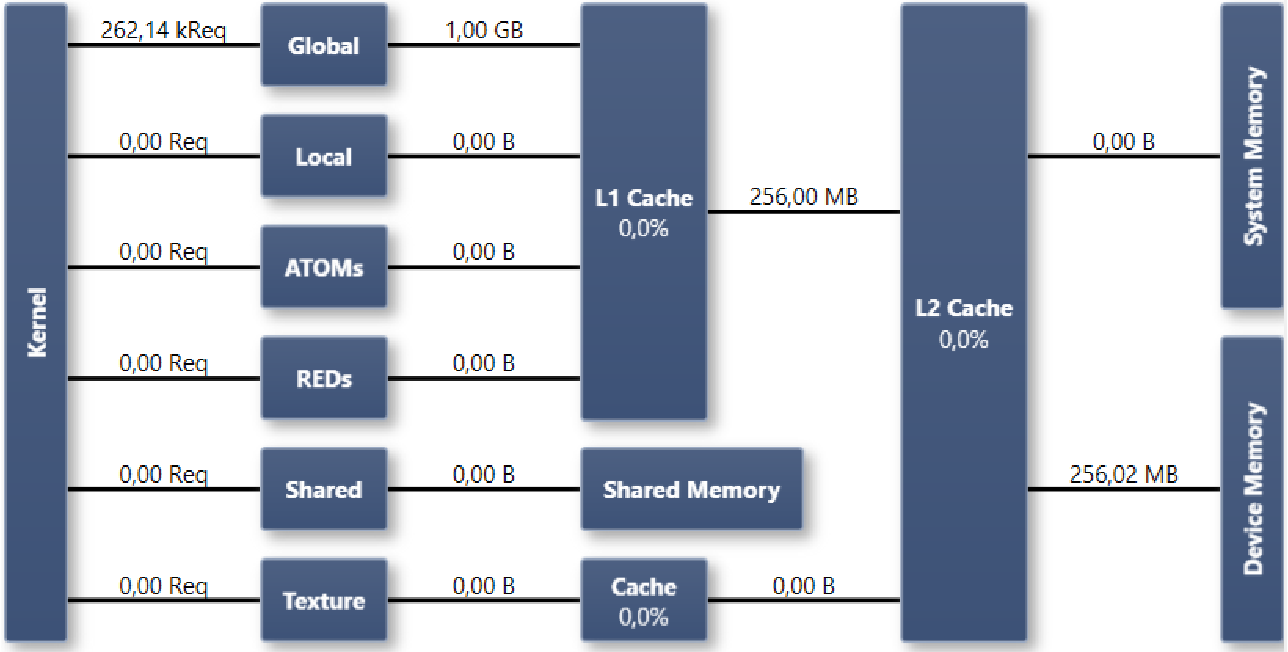
Offset 32 6.54GB/s (0.09)

Granularity Example 4/4

```
__global__ void testScattered(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];
}
```

```
testCoalesced done in 0.240736ms -> 69.6914GB/s
testOffset<32> done in 2.5617ms -> 6.54351GB/s
testScattered done in 4.70234ms -> 3.56785GB/s
```

Granularity Example 4/4



Scattered 3.57GB/s (0.05)

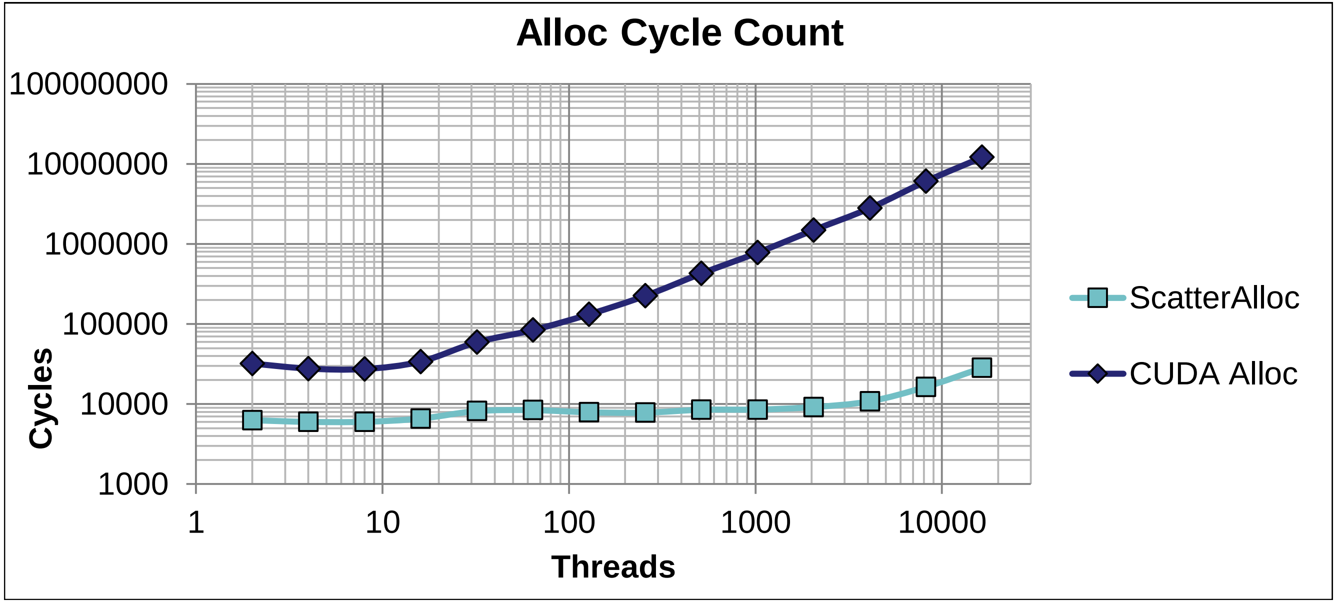
Granularity Example Summary

testCoalesced done in	0.2407ms	->	69.6914GB/s
testMixed done in	0.2478ms	->	67.6937GB/s
testOffset<2> done in	0.4111ms	->	40.8073GB/s
testOffset<4> done in	0.8276ms	->	20.2717GB/s
testOffset<8> done in	1.6669ms	->	10.0647GB/s
testOffset<32> done in	2.5617ms	->	6.54351GB/s
testScattered done in	4.7023ms	->	3.56785GB/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

- malloc / free, new / delete are possible for cc >= 2.0
- 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 64KB
- Fermi added uniform access:
  - Load Uniform (LDU) command for const pointer
  - Goes through same cache
  - Compiler determines that all threads in a block will dereference the same address
  - no size limit

Constant memory cont'd

- 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];
__device__ float d_array[128];
__device__ const float dc_array[128];
__global__ void kernel()
{
  float a = c_array[0];             // ld.const     25ns
  float b = c_array[blockIdx.x];    // ld.const     25ns
  float c = c_array[threadIdx.x];   // ld.const    920ns
  float d = d_array[blockIdx.x];    // ld.global    21ns
  float e = d_array[threadIdx.x];   // ld.global    42ns
  float f = dc_array[blockIdx.x];   // ldu          21ns
  float g = dc_array[threadIdx.x];// ld.global    42ns
}
```

Constant Memory Example 2

```
__constant__ float c_array[1024];
__device__ float d_array[1024];
__device__ const float dc_array[1024];
__global__ void kernel()
{
 for(int i = 0; i < 800; i+= 8)
 {
   float a = c_array[i];              // ld.const     25ns
   float b = c_array[blockIdx.x+i];   // ld.const     25ns
   float c = c_array[threadIdx.x+i];  // ld.const   1195ns
   float d = d_array[blockIdx.x+i];   // ld.global    84ns
   float e = d_array[threadIdx.x+i];  // ld.global   127ns
   float f = dc_array[blockIdx.x+i];  // ldu         84ns
   float g = dc_array[threadIdx.x+i];// ld.global   127ns
 }
}
```

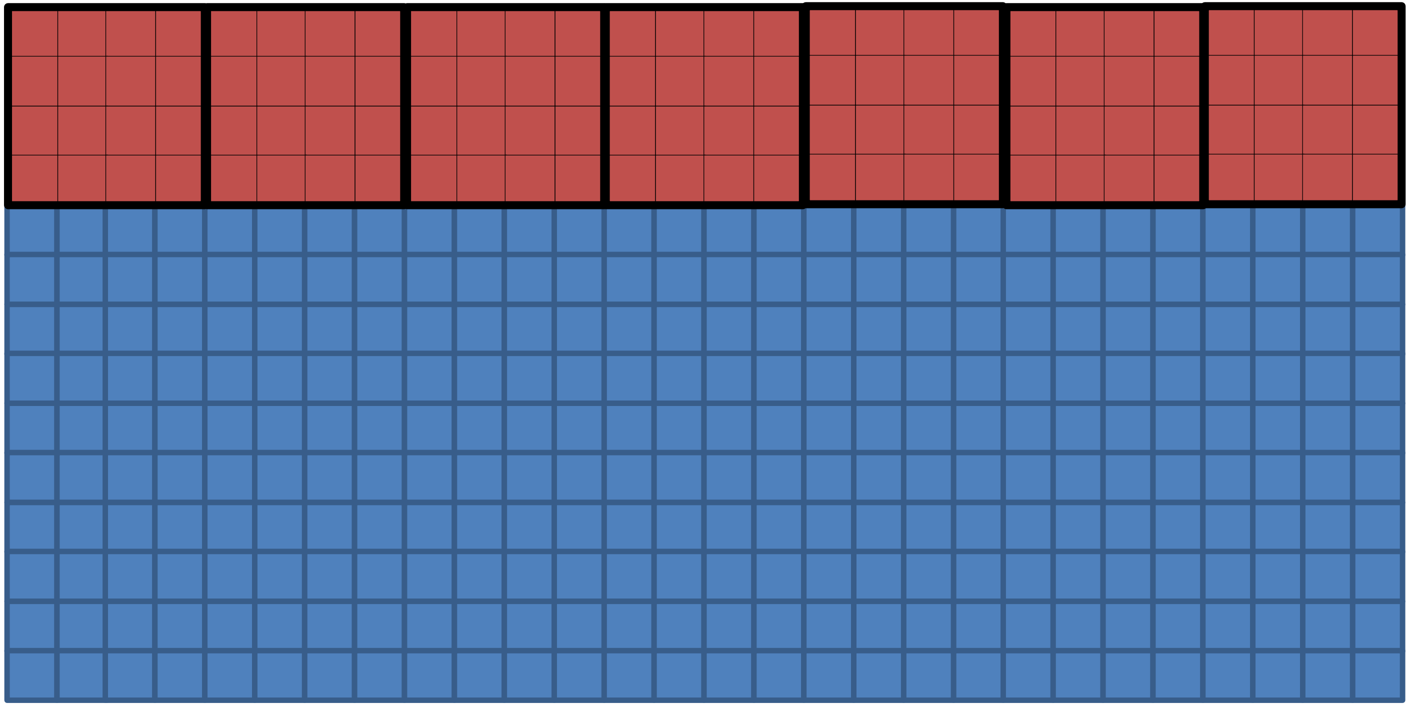
Constant memory

- Only useable 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

- Separate cache
- 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
- Can also bind global memory to textures

RowAfterRow

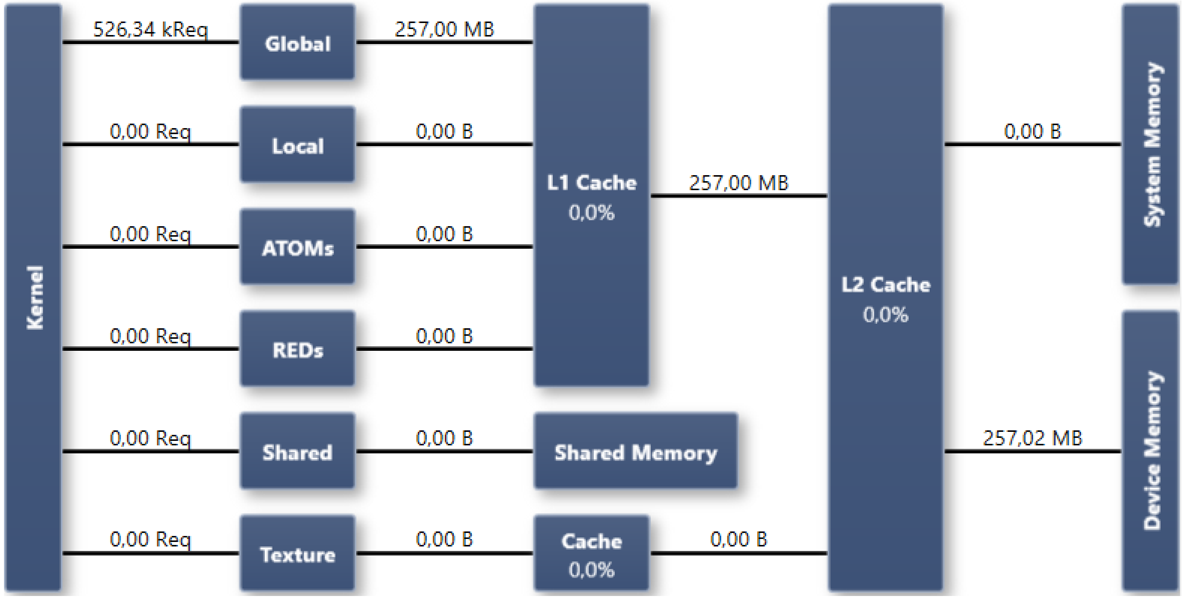


RowAfterRow: Texture vs Global

```
__global__ void deviceLoadRowAfterRow(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;
}
```

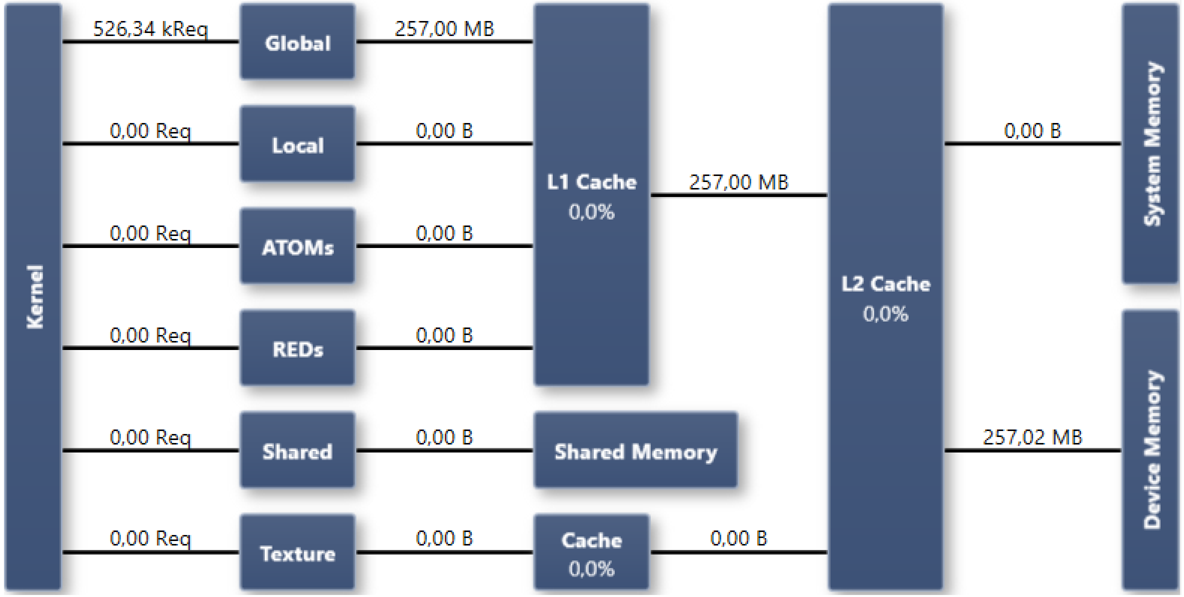
```
__global__ void texLoadRowAfterRow( int width, int height, float4* out)
{
  float4 sum = make_float4(0,0,0,0);
  int xin = blockIdx.x*blockDim.x + threadIdx.x;
  int yin = blockIdx.y*blockDim.y + threadIdx.y;
  for(int y = yin; y < height; y+=blockDim.y)
  {
    float4 in = tex2D(myTex,xin,y);
    sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
  }
  out[xin + yin*gridDim.x*blockDim.x] = sum;
}
```

RowAfterRow: Texture vs Global



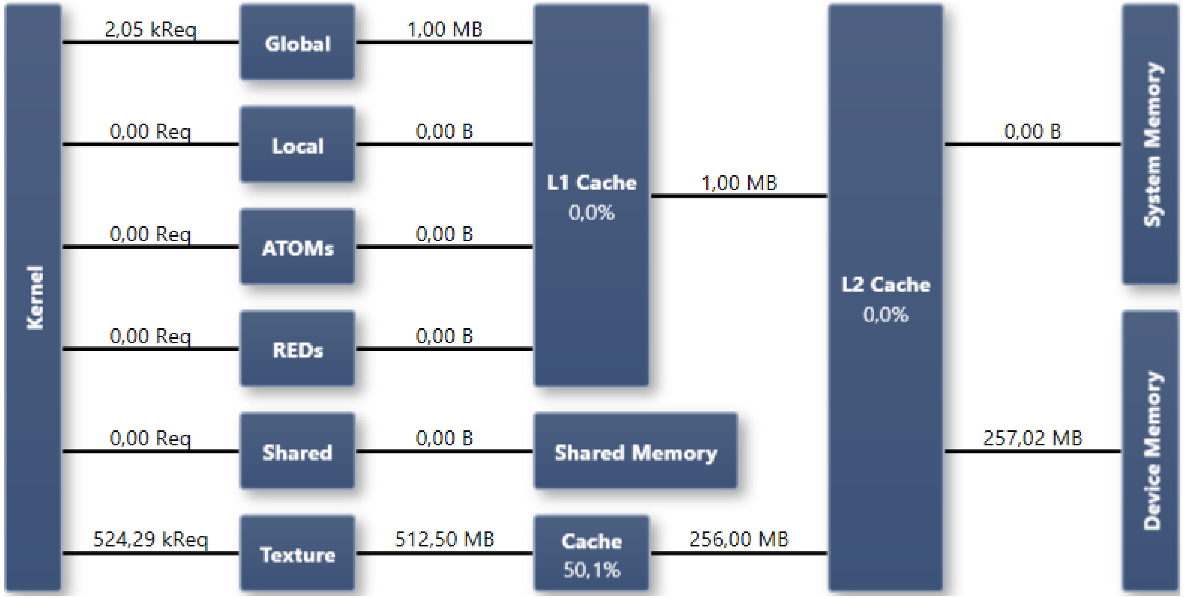
global linear (cudaMalloc) 177GB/s

RowAfterRow: Texture vs Global



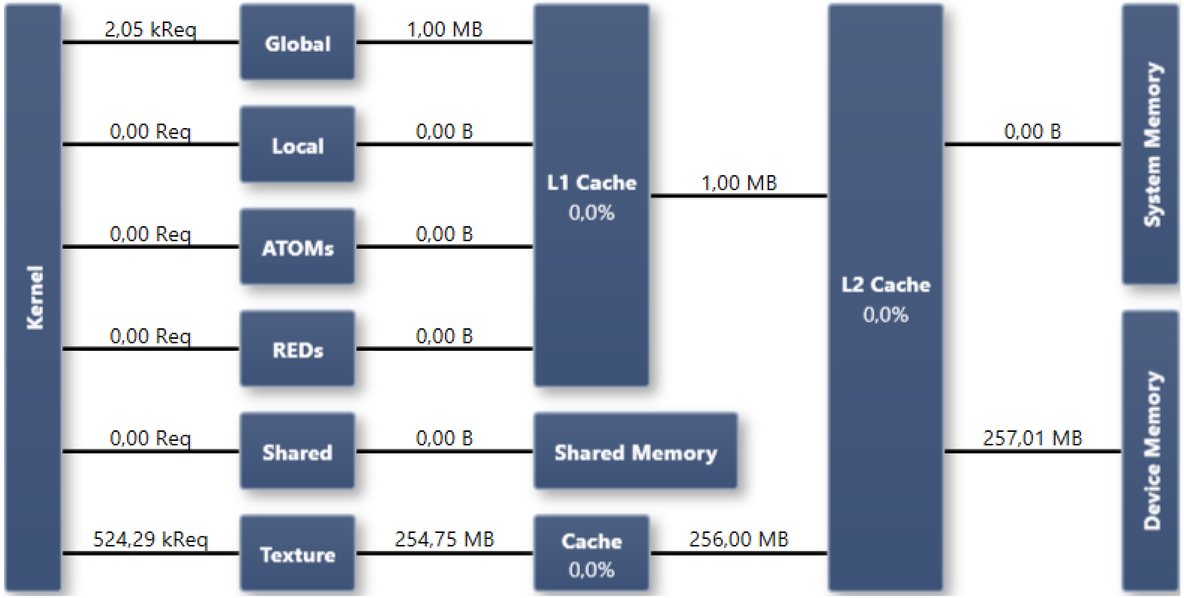
global 2D (cudaMallocPitch) 178GB/s

RowAfterRow: Texture vs Global



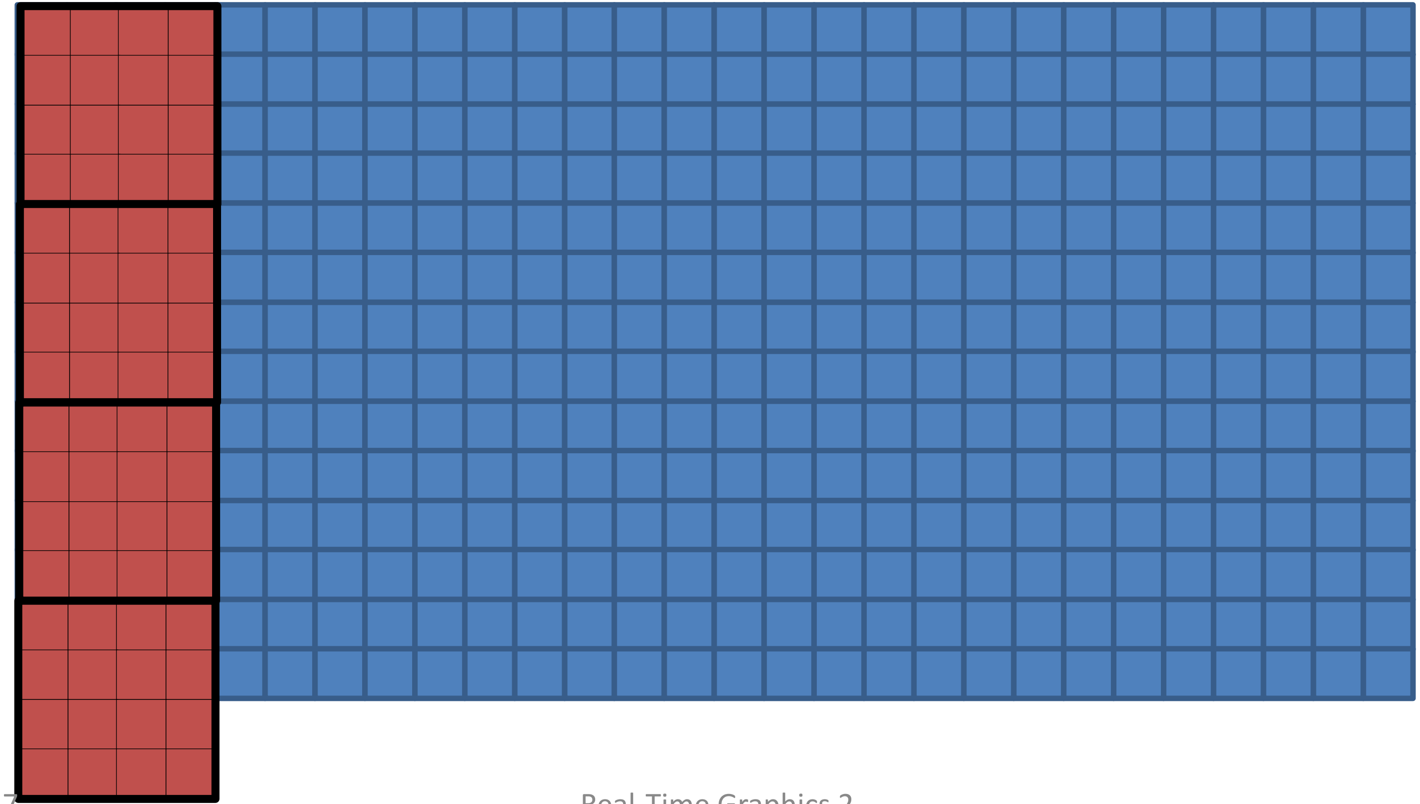
texture (cudaArray) 174GB/s

RowAfterRow: Texture vs Global



texture (cudaMallocPitch) 175GB/s

ColumnAfterColumn

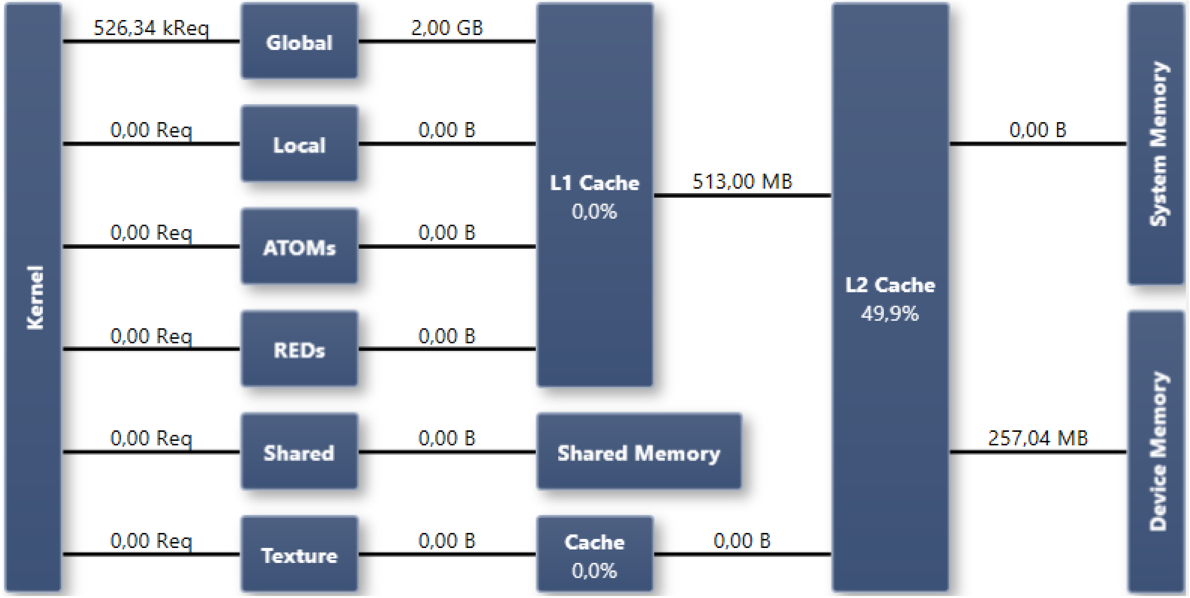


ColumnAfterColumn: Texture vs Global

```
__global__ void deviceLoadColumnAfterColumn(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 x = xin; x < width; x+=blockDim.x)
  {
    float4 in = data[yin + x*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;
}
```

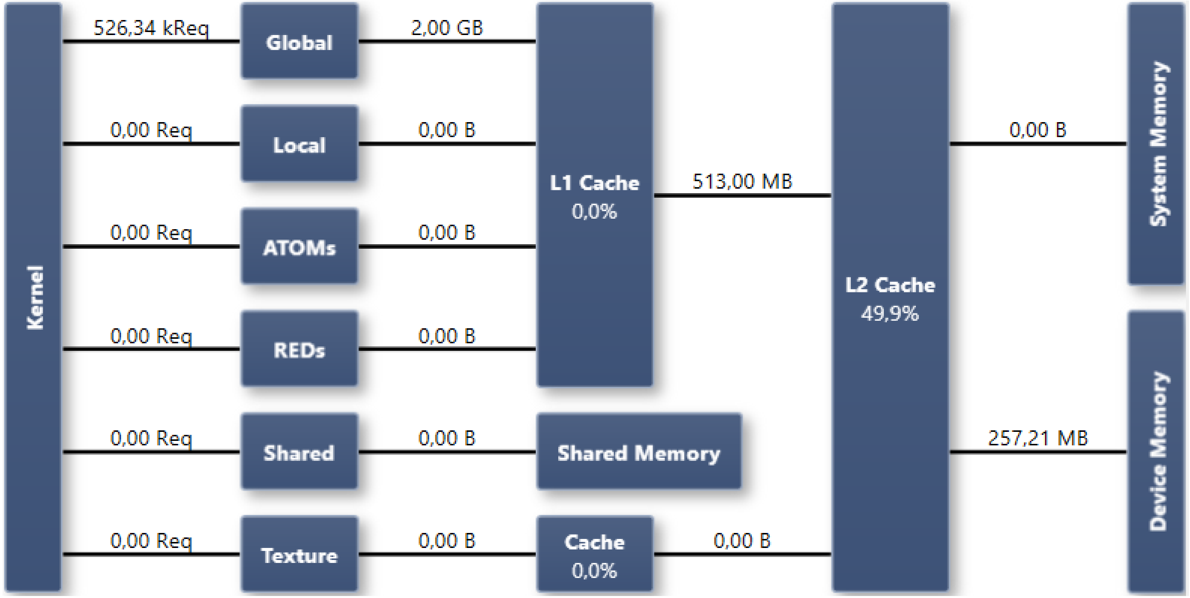
```
__global__ void texLoadColumnAfterColumn( int width, int height, float4* out)
{
  float4 sum = make_float4(0,0,0,0);
  int xin = blockIdx.x*blockDim.x + threadIdx.x;
  int yin = blockIdx.y*blockDim.y + threadIdx.y;
  for(int x = xin; x < width; x+=blockDim.x)
  {
    float4 in = tex2D(myTex,yin,x);
    sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
  }
  out[xin + yin*gridDim.x*blockDim.x] = sum;
}
```

ColumnAfterColumn: Texture vs Global



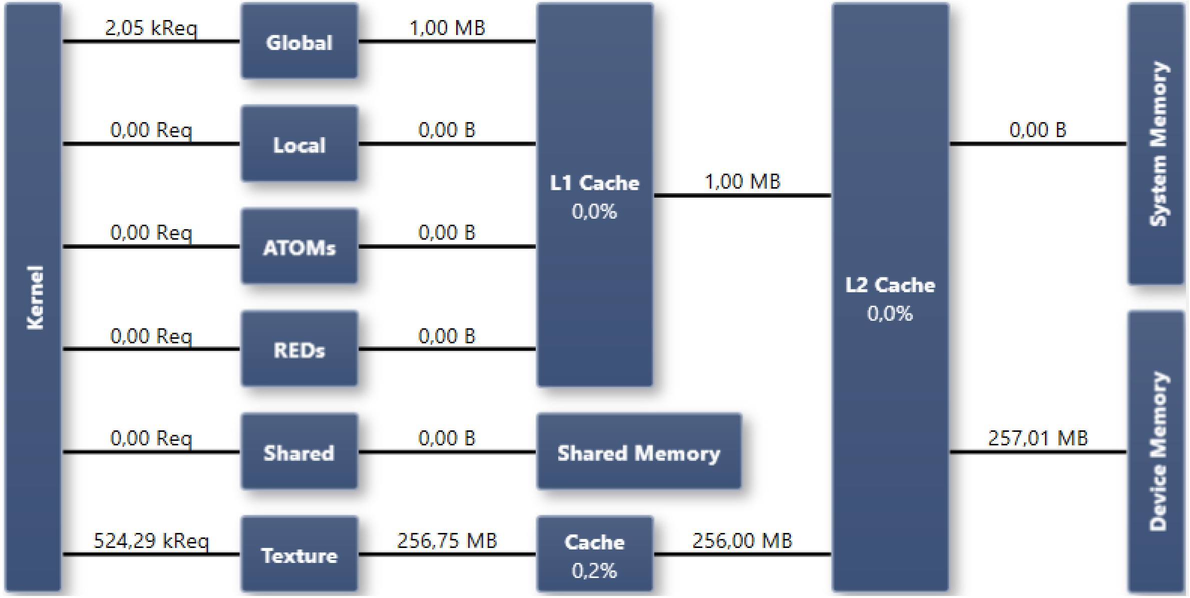
global linear (cudaMalloc) 46GB/s

ColumnAfterColumn: Texture vs Global



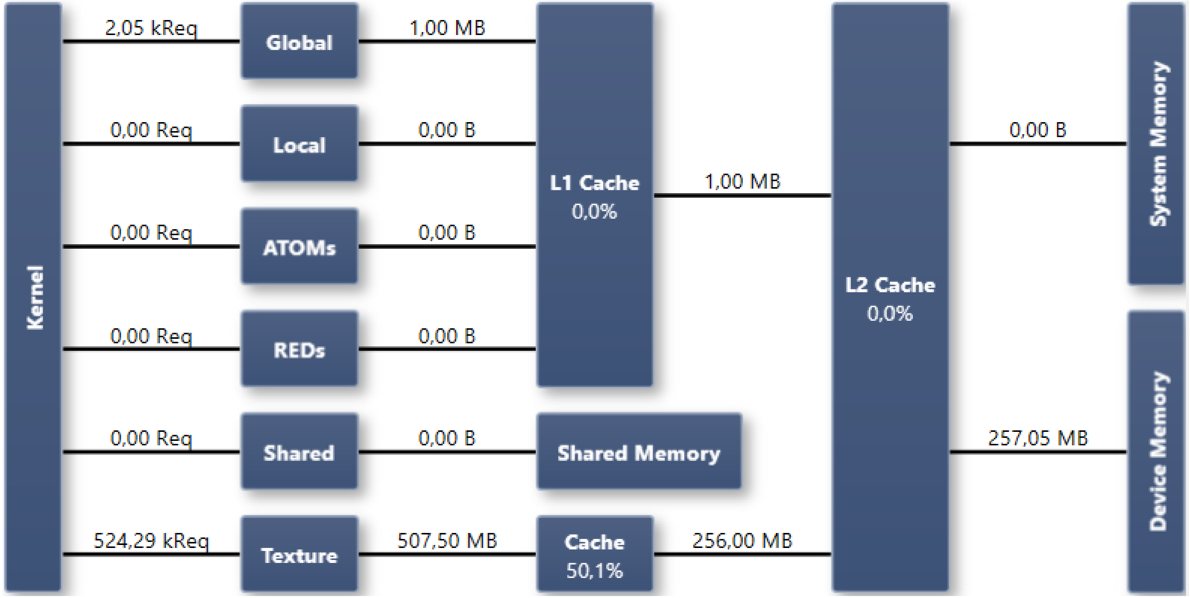
global 2D (cudaMallocPitch) 46GB/s

ColumnAfterColumn: Texture vs Global



texture (cudaArray) 171GB/s

ColumnAfterColumn: Texture vs Global



texture (cudaMallocPitch) 63GB/s

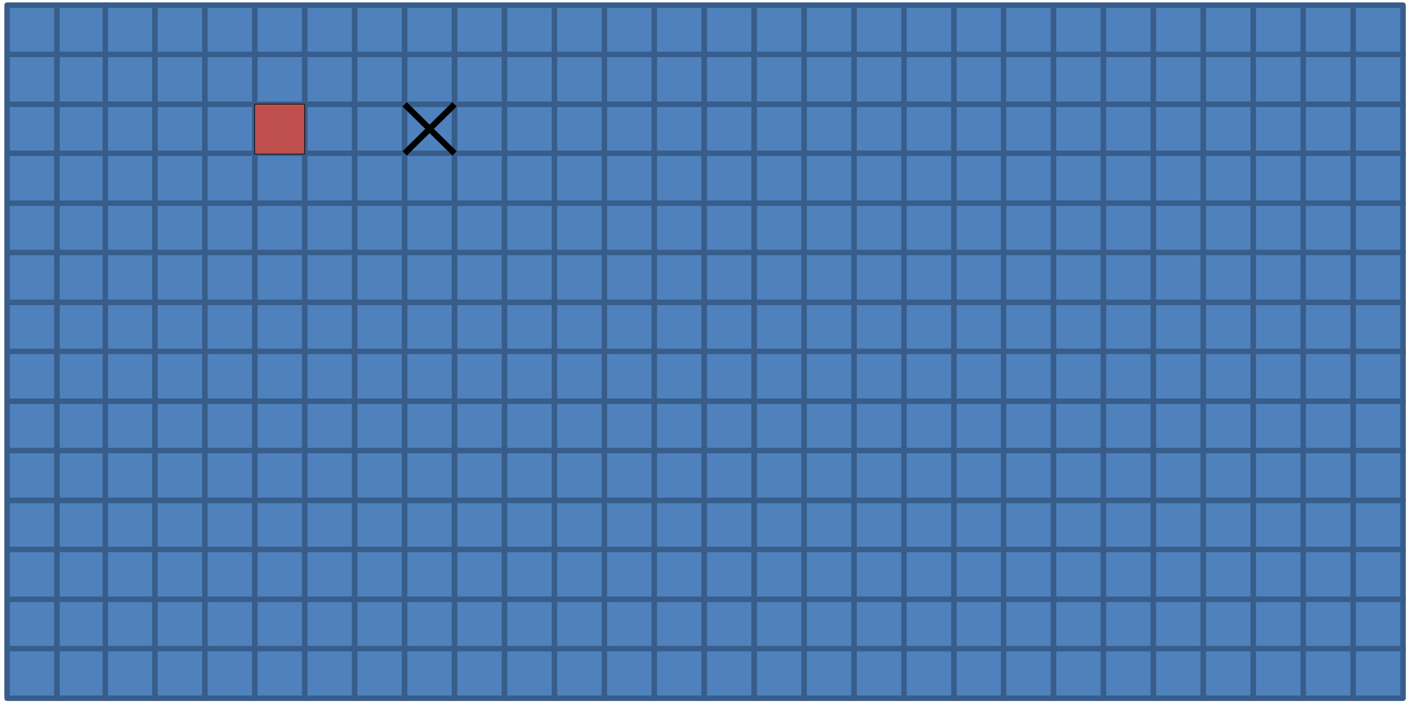
Texture vs Global: Sliding

globalLinRowAfterRow	1.50963ms	-> 177.815GB/s
global2DRowAfterRow	1.50355ms	-> 178.534GB/s
texArrayRowAfterRow	1.53712ms	-> 174.635GB/s
tex2DMemRowAfterRow	1.53299ms	-> 175.106GB/s

globalLinColumnAfterColumn	5.85152ms	-> 45.8745GB/s
global2DColumnAfterColumn	5.85245ms	-> 45.8672GB/s
texArrayColumnAfterColumn	1.56563ms	-> 171.455GB/s
tex2DMemColumnAfterColumn	4.25344ms	-> 63.1102GB/s

- Row access is similarly efficient with all types
- Column access via cudaArray allows for direct load while others need cache
- Texture access more efficient than L2 cache

Filter X



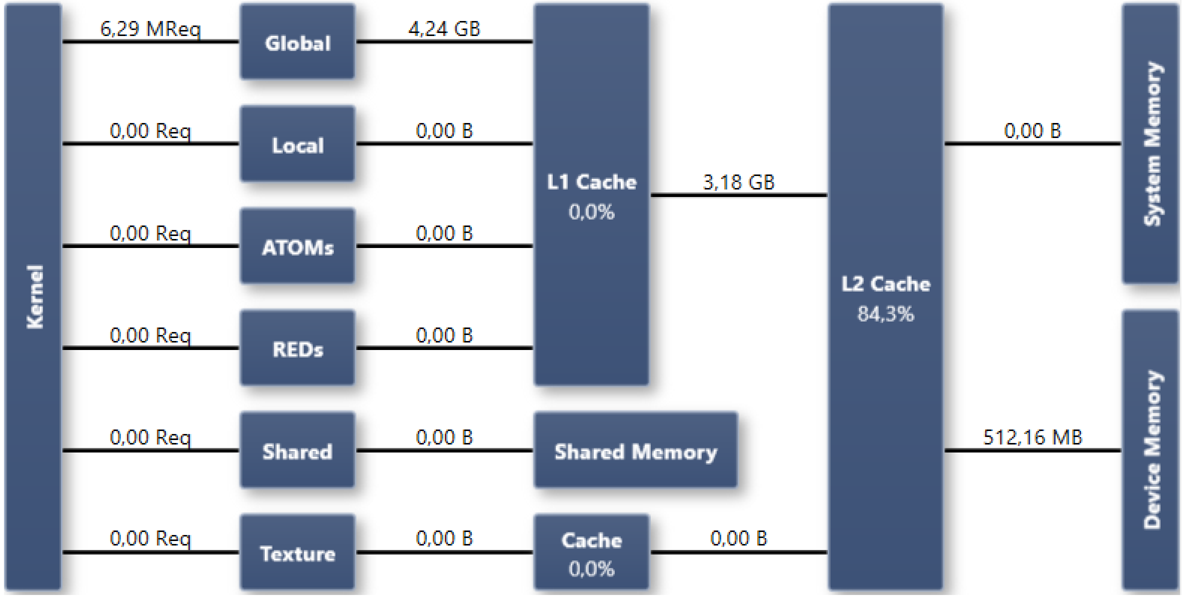
FilterX: Texture vs Global

```
__global__ void globalFilterX(const float4* data, int pitch, int width, int height, float4* out, int offset)
{
  float4 sum = make_float4(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)
      {
        float4 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;
}
```

FilterX: Texture vs Global

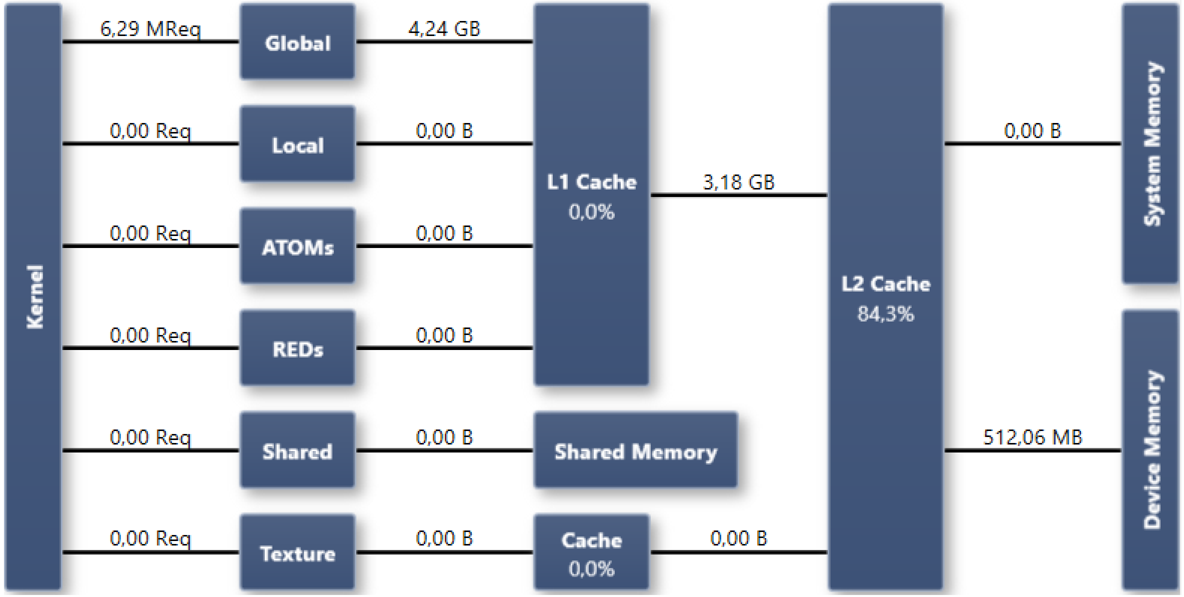
```
__global__ void texFilterX(int width, int height, float4* out, int offset)
{
  float4 sum = make_float4(0,0,0,0);
  int xin = blockIdx.x*blockDim.x + threadIdx.x;
  int yin = blockIdx.y*blockDim.y + threadIdx.y;

  for(int x = xin-offset; x <= xin+offset; ++x)
  {
    float4 in = tex2D(myTex,x,yin);
    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;
}
```

FilterX: Texture vs Global



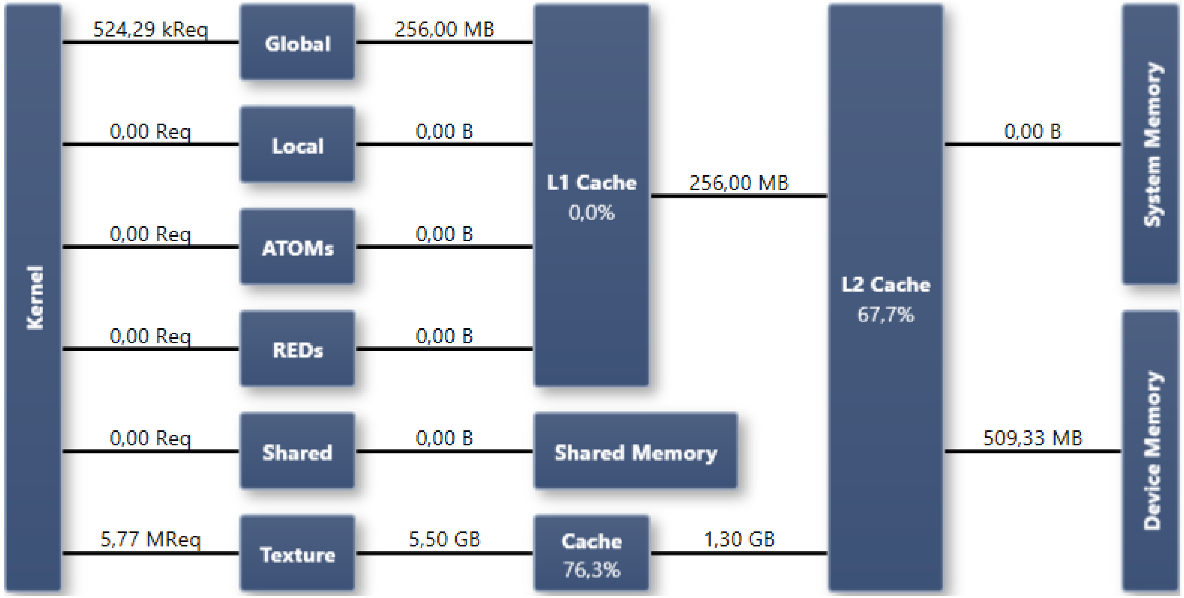
global linear (cudaMalloc) 273GB/s

FilterX: Texture vs Global



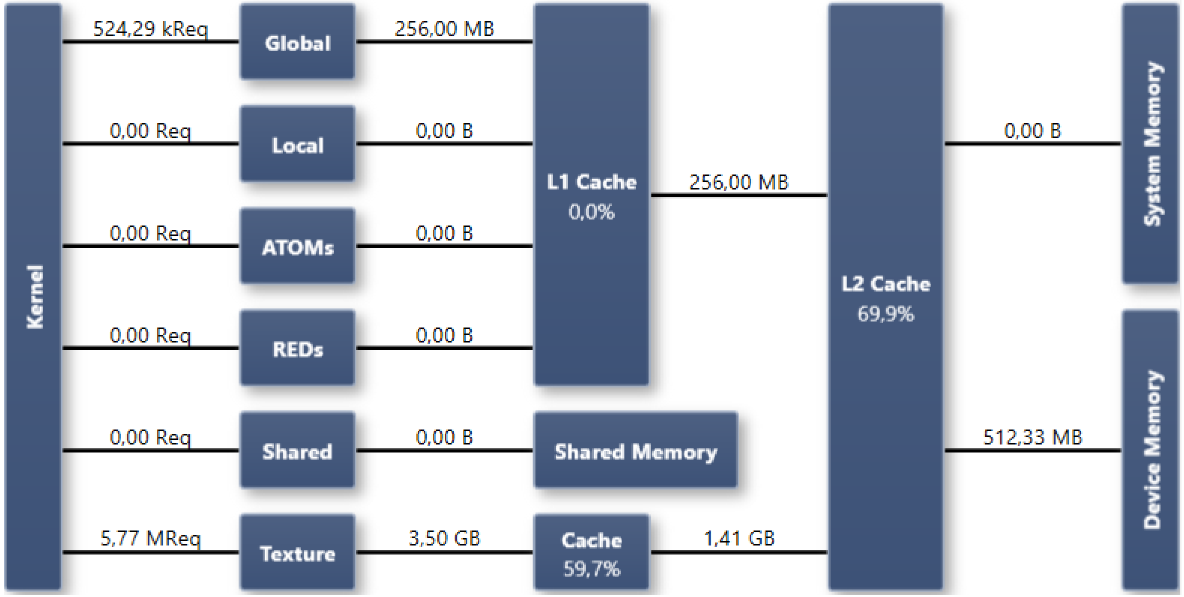
global 2D (cudaMallocPitch) 274GB/s

FilterX: Texture vs Global



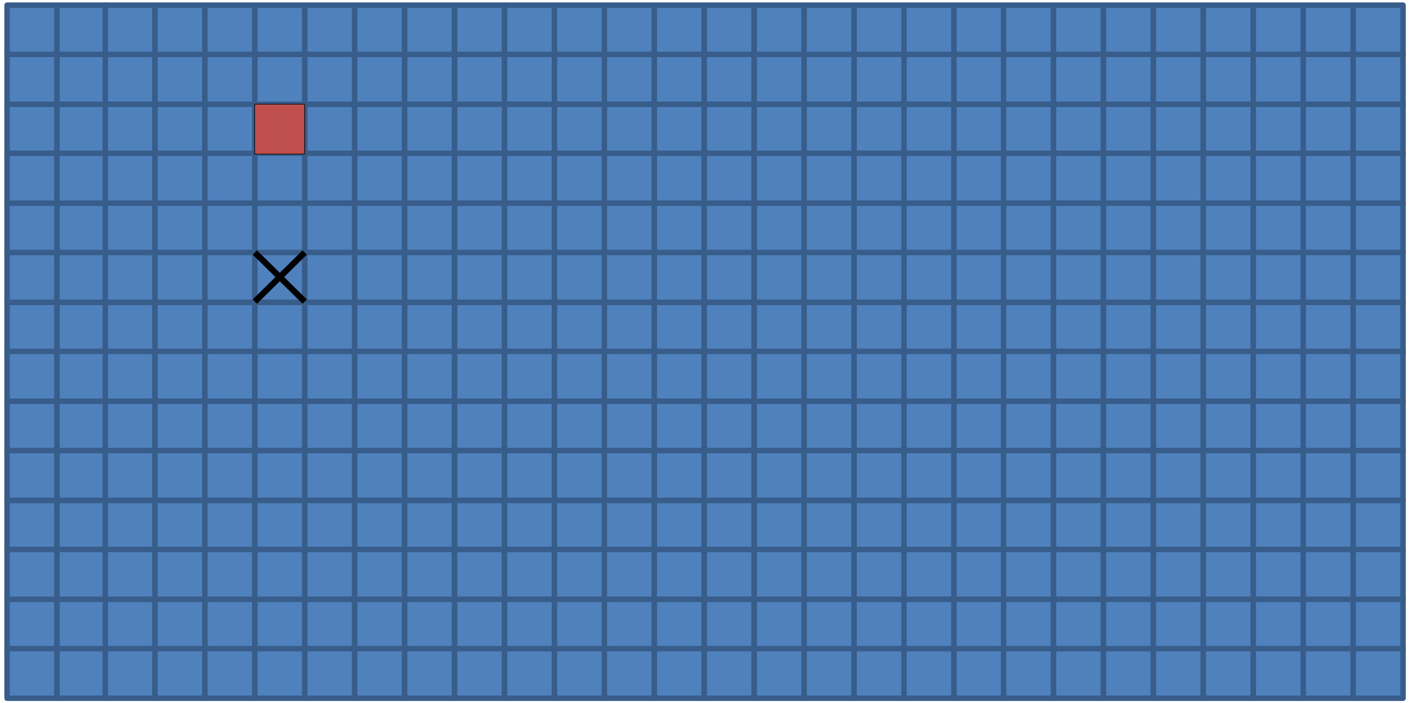
texture (cudaArray) 587GB/s

FilterX: Texture vs Global



texture (cudaMallocPitch) 514GB/s

Filter Y



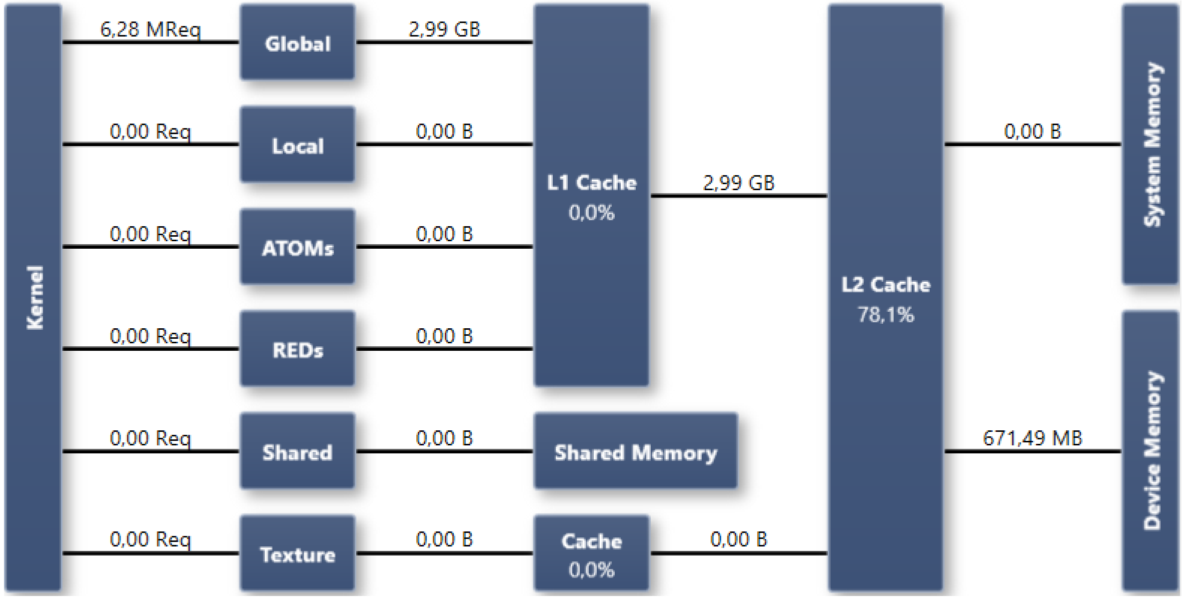
FilterY: Texture vs Global

```
__global__ void globalFilterY(const float4* data, int pitch, int width, int height, float4* out, int offset)
{
  float4 sum = make_float4(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)
      {
        float4 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;
}
```

FilterY: Texture vs Global

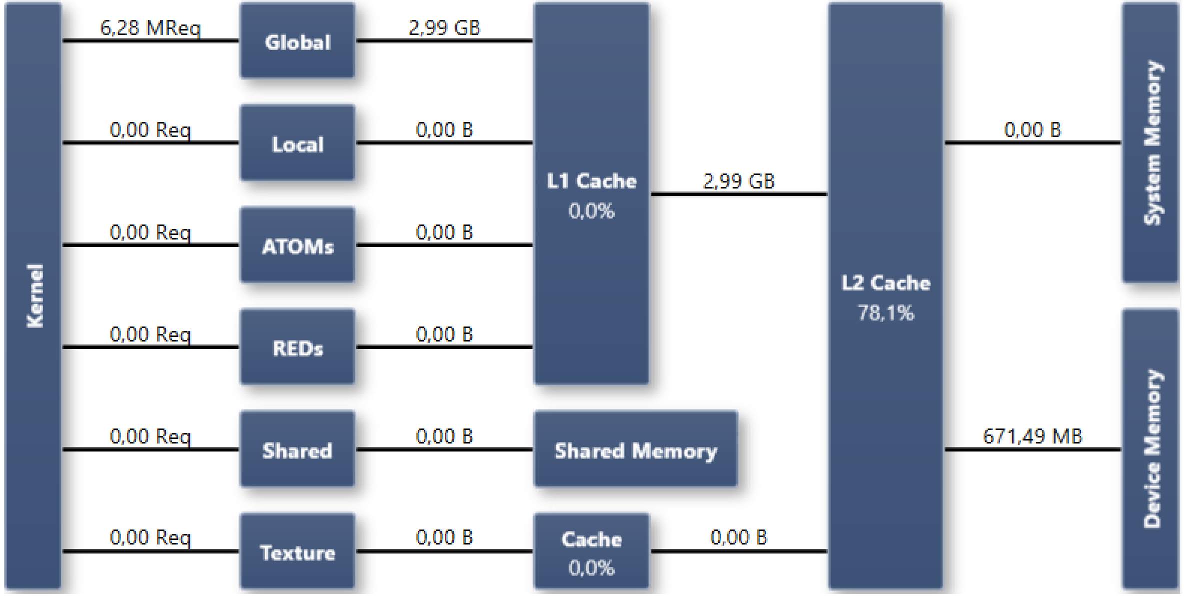
```
__global__ void texFilterY(int width, int height, float4* out, int offset)
{
  float4 sum = make_float4(0,0,0,0);
  int xin = blockIdx.x*blockDim.x + threadIdx.x;
  int yin = blockIdx.y*blockDim.y + threadIdx.y;

  for(int y = yin-offset; y <= yin+offset; ++y)
  {
    float4 in = tex2D(myTex,xin,y);
    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;
}
```

FilterY: Texture vs Global



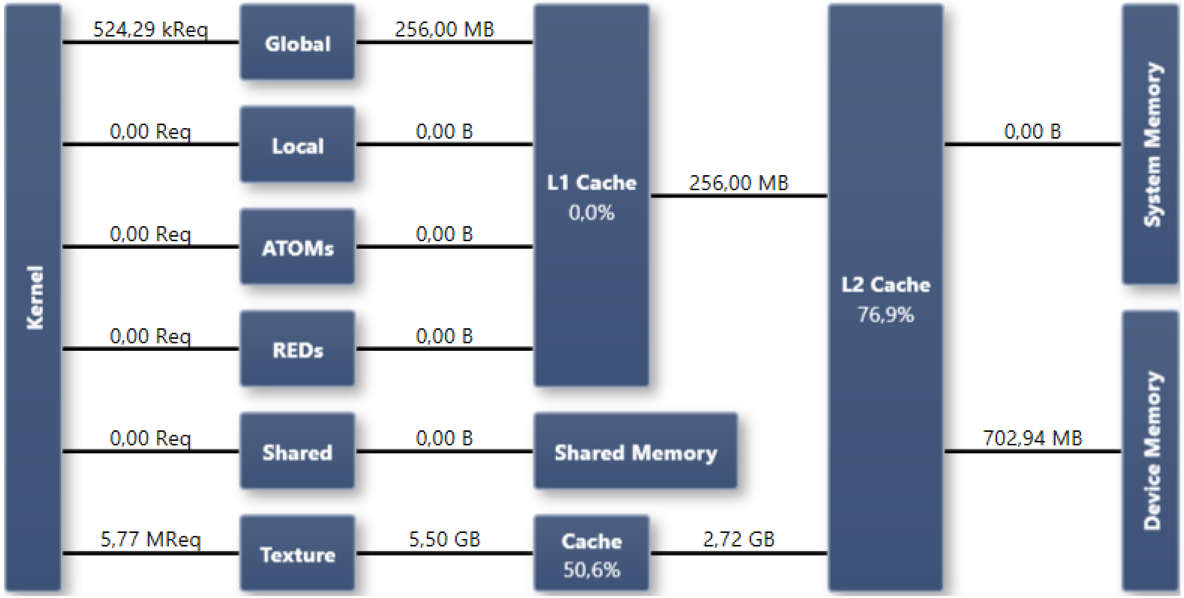
global linear (cudaMalloc) 291GB/s

FilterY: Texture vs Global



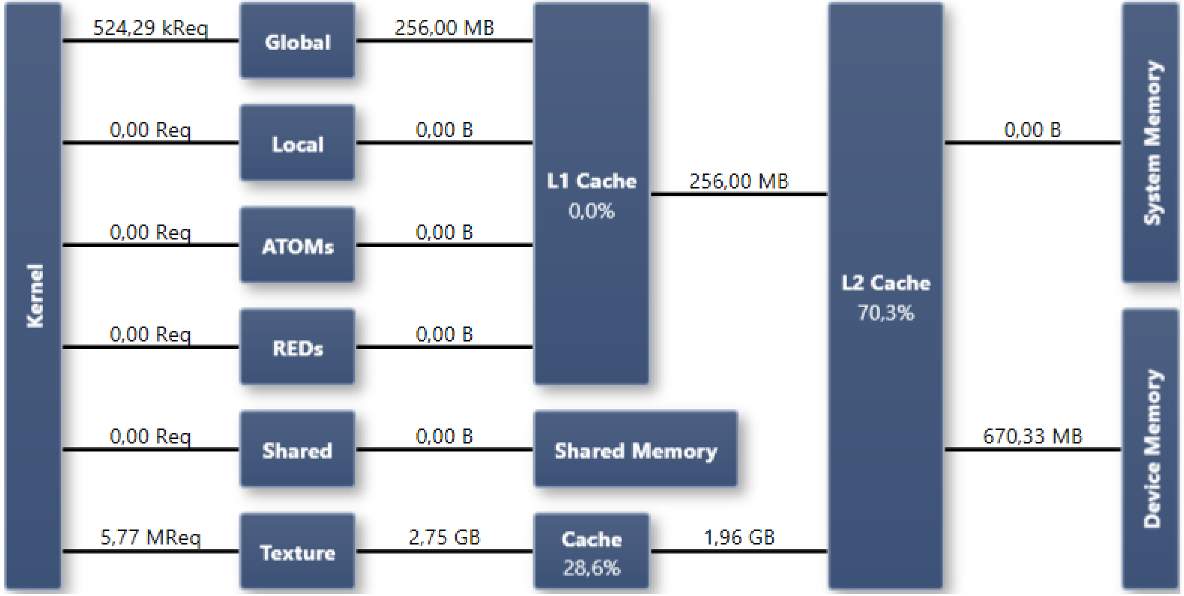
global 2D (cudaMallocPitch) 291GB/s

FilterY: Texture vs Global



texture (cudaArray) 364GB/s

FilterY: Texture vs Global



texture (cudaMallocPitch) 495GB/s

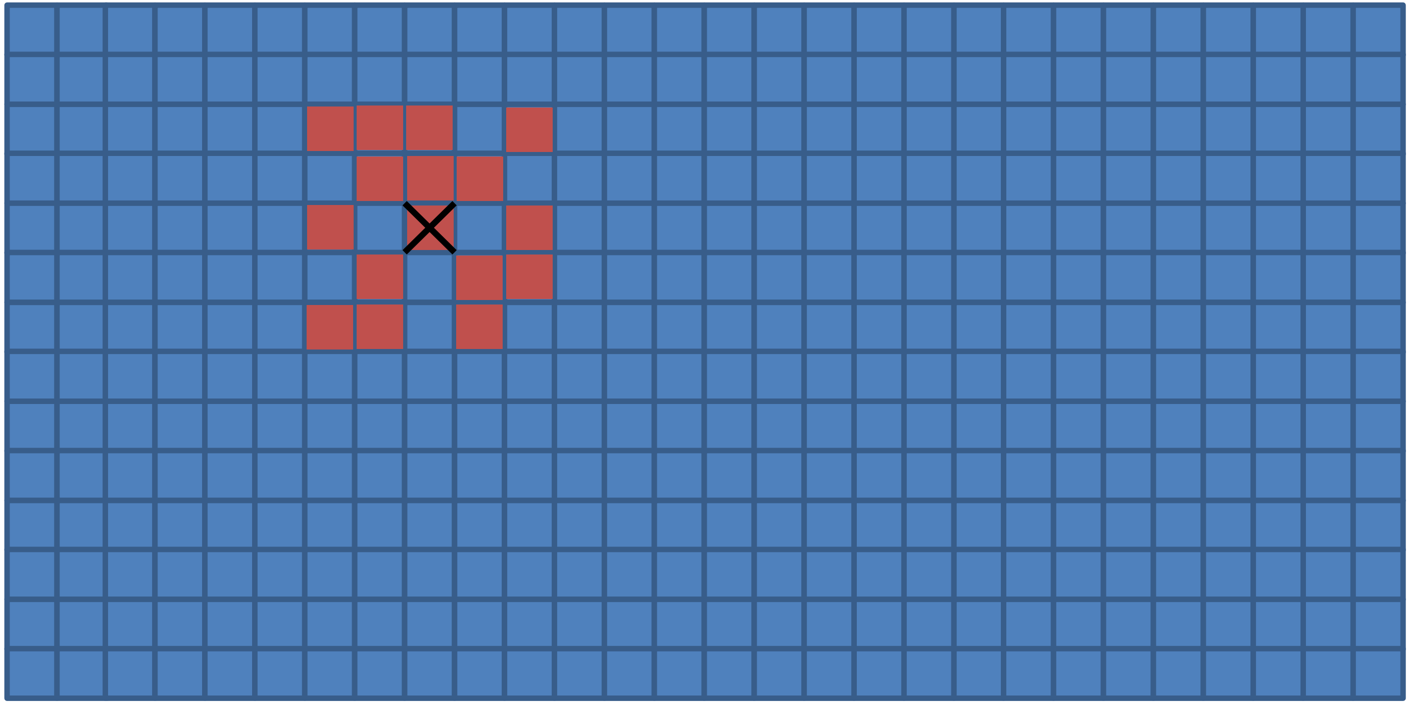
Texture vs Global: Filtering

globalLinFilterX	10.7816ms -> 273.872GB/s
global2DFilterX	10.7704ms -> 274.157GB/s
texArrayFilterX	5.02566ms -> 587.542GB/s
tex2DFilterX	5.73747ms -> 514.65GB/s

globalLinFilterY	10.1444ms -> 291.076GB/s
global2DFilterY	10.1401ms -> 291.2GB/s
texArrayFilterY	8.09706ms -> 364.674GB/s
tex2DFilterY	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



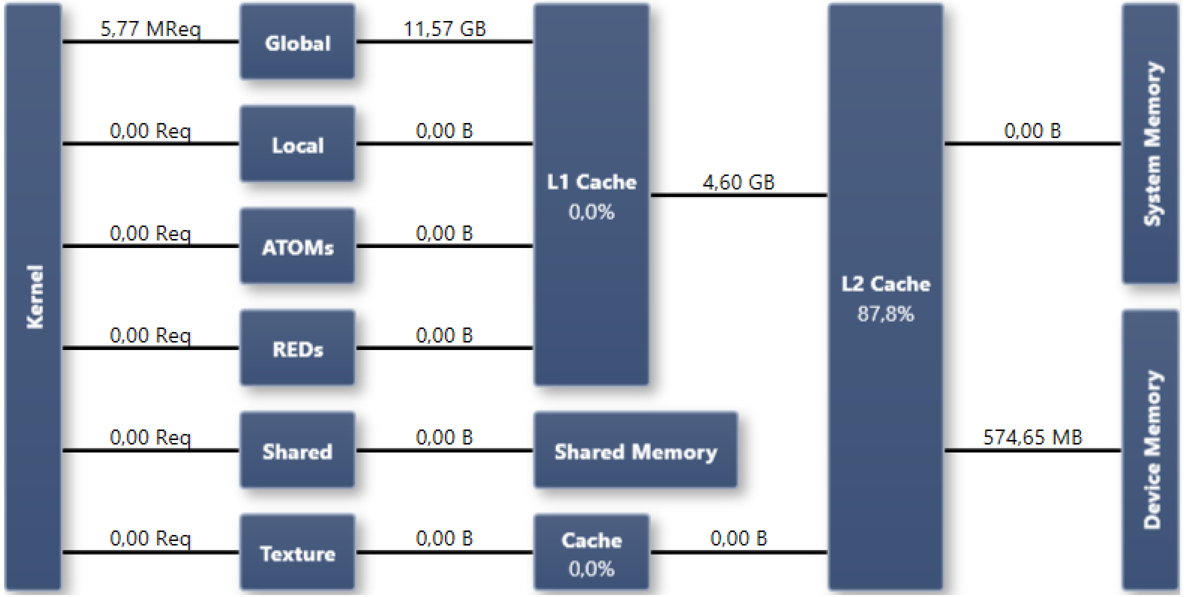
Random Sampling: Texture vs Global

```
template<int Area>
__global__ void globalRandomSample(const float4* data, int pitch, int width, int height, float4* out, int samples)
{
  float4 sum = make_float4(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);
    float4 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

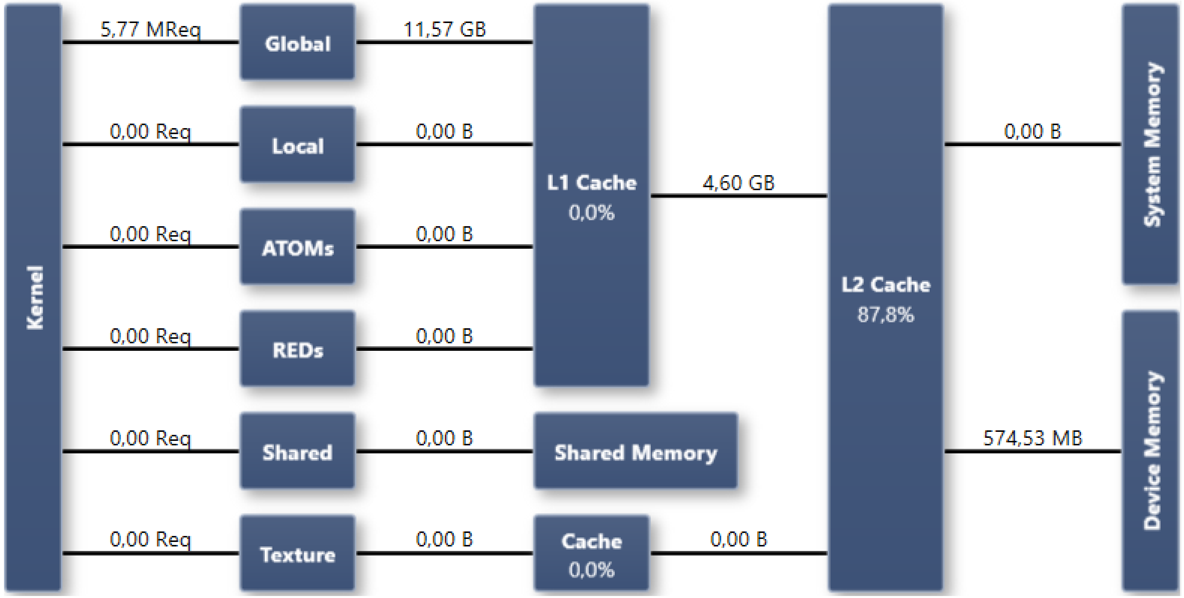
```
template<int Area>
__global__ void texRandomSmple(int width, int height, float4* out, int samples)
{
  float4 sum = make_float4(0,0,0,0);
  int xin = blockIdx.x*blockDim.x + Area*(threadIdx.x/Area);
  int yin = blockIdx.y*blockDim.y + Area*(threadIdx.y/Area);

  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);
    float4 in = tex2D(myTex, xin + x, yin + y);
    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



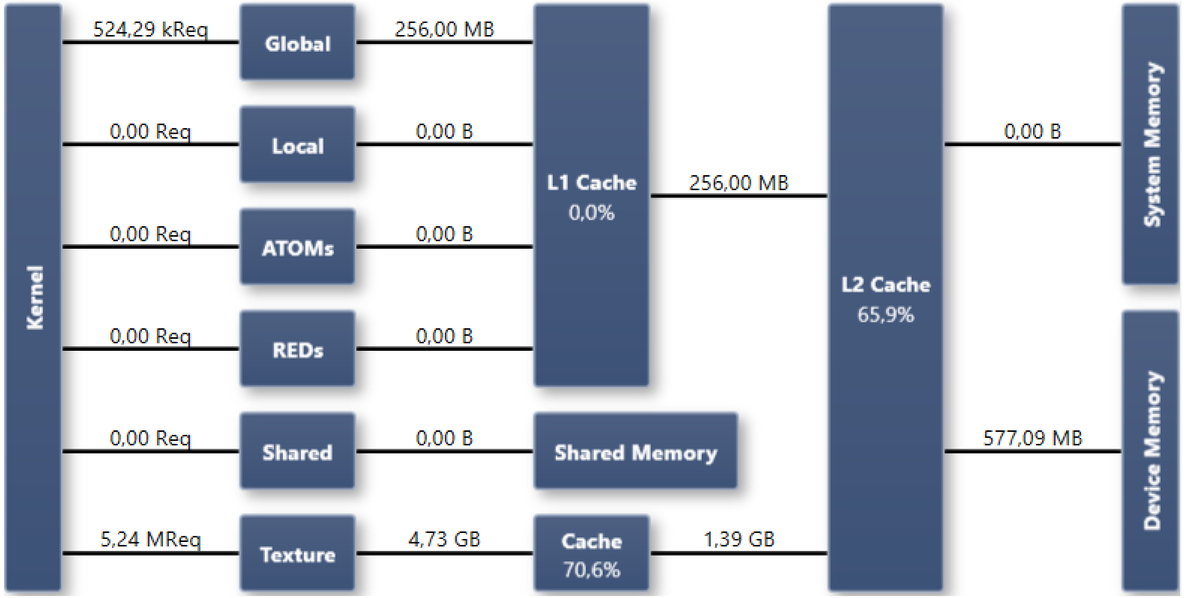
global linear (cudaMalloc) 112GB/s

Random Sampling: Texture vs Global



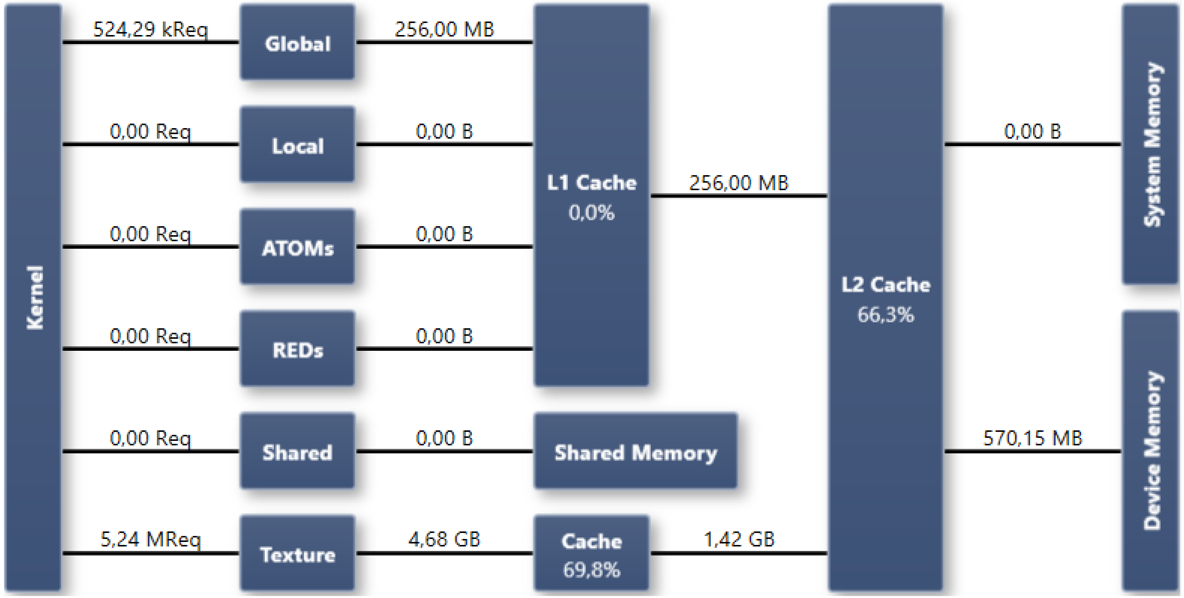
global 2D (cudaMallocPitch) 112GB/s

Random Sampling: Texture vs Global



texture (cudaArray) 355GB/s

Random Sampling: Texture vs Global



texture (cudaMallocPitch) 356GB/s

Texture vs Global: Random Sampling

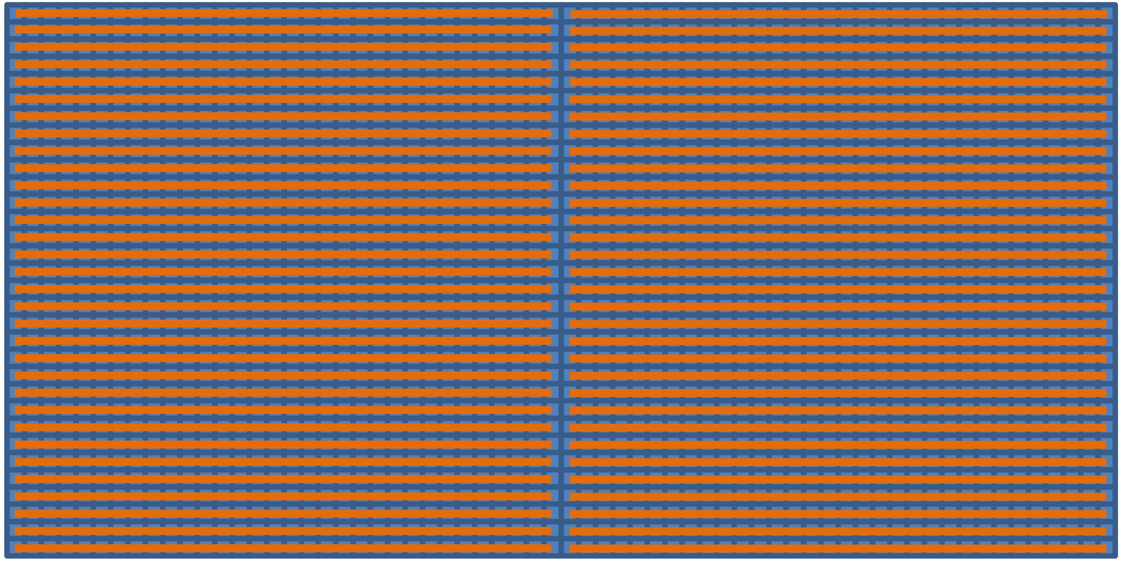
globalLinRandomSample	23.9213ms -> 112.216GB/s
global2DRandomSample	23.9196ms -> 112.224GB/s
texArrayRandomSample	7.54998ms -> 355.544GB/s
tex2DRandomSample	7.53933ms -> 356.047GB/s

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

Texture vs Global Conclusion

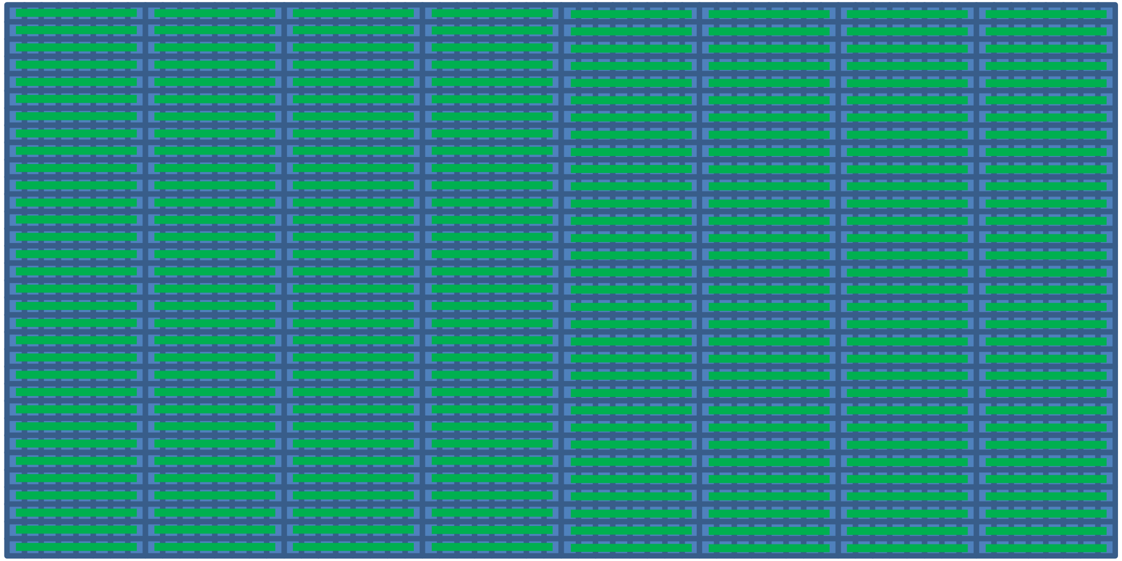
- Textures always perform as good or better
  - With textures performance less dependent on access direction/pattern
  - Textures put less stress on L2 cache
  - Texture cache separate from L1 cache (L1 free for other tasks)
  - Features: border handling, interpolation, conversion

Array vs Pitched Textures



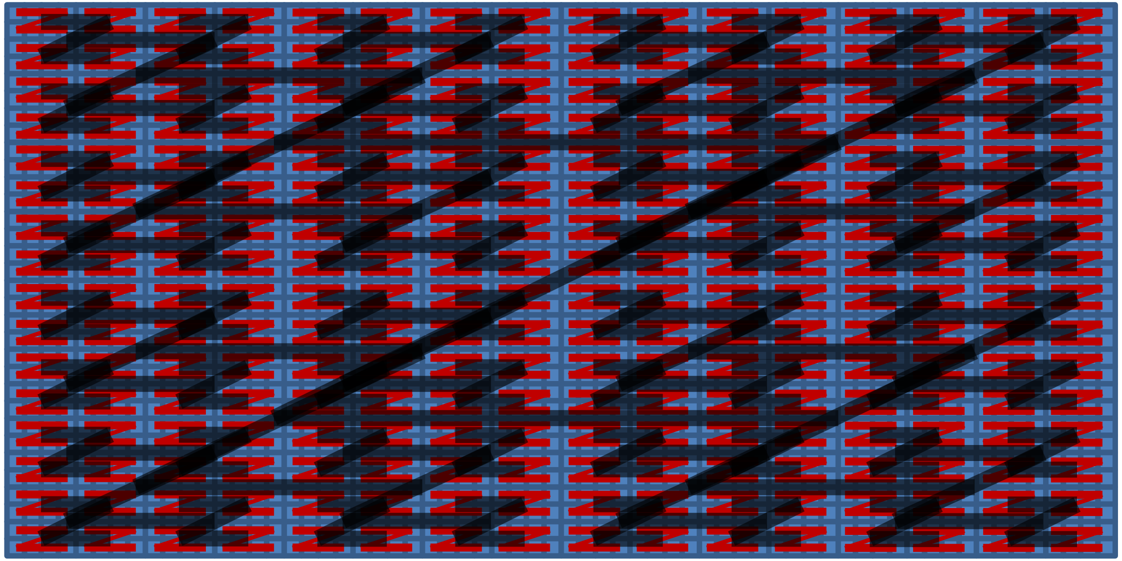
- Pitched global access: cache line size 128 byte

Array vs Pitched Textures



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

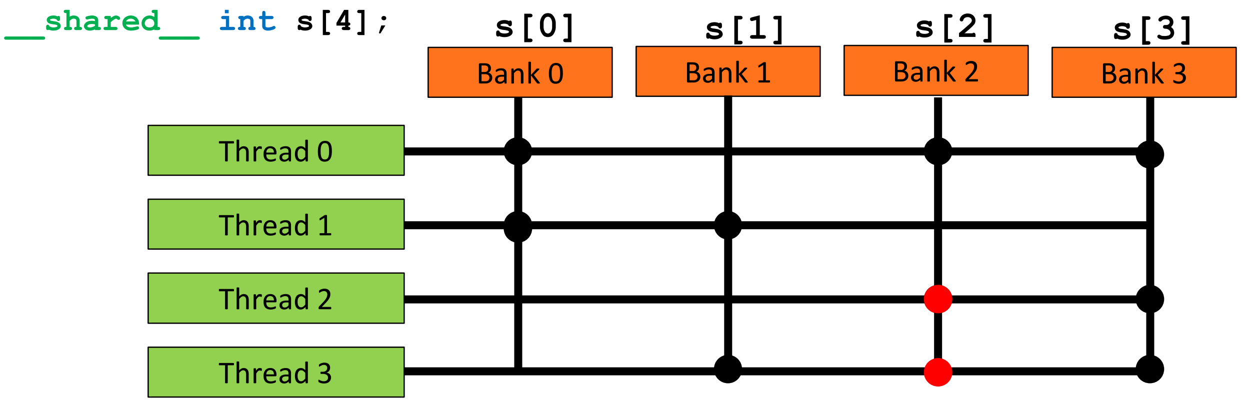
Array vs Pitched Textures



- 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 memory

- Accessed via crossbar → access pattern important
- Different behavior for different architectures
  - Pre-Fermi:
    access issued per 16 threads
    16 banks a 32 bits per 2 clock cycles

    bank = (address/4) % 16
  - Fermi:
    access issued per 32 threads
    32 banks a 32 bits per 2 clock cycles

    bank = (address/4) % 32
  - Kepler:
    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
    modes can be set using the CUDA API

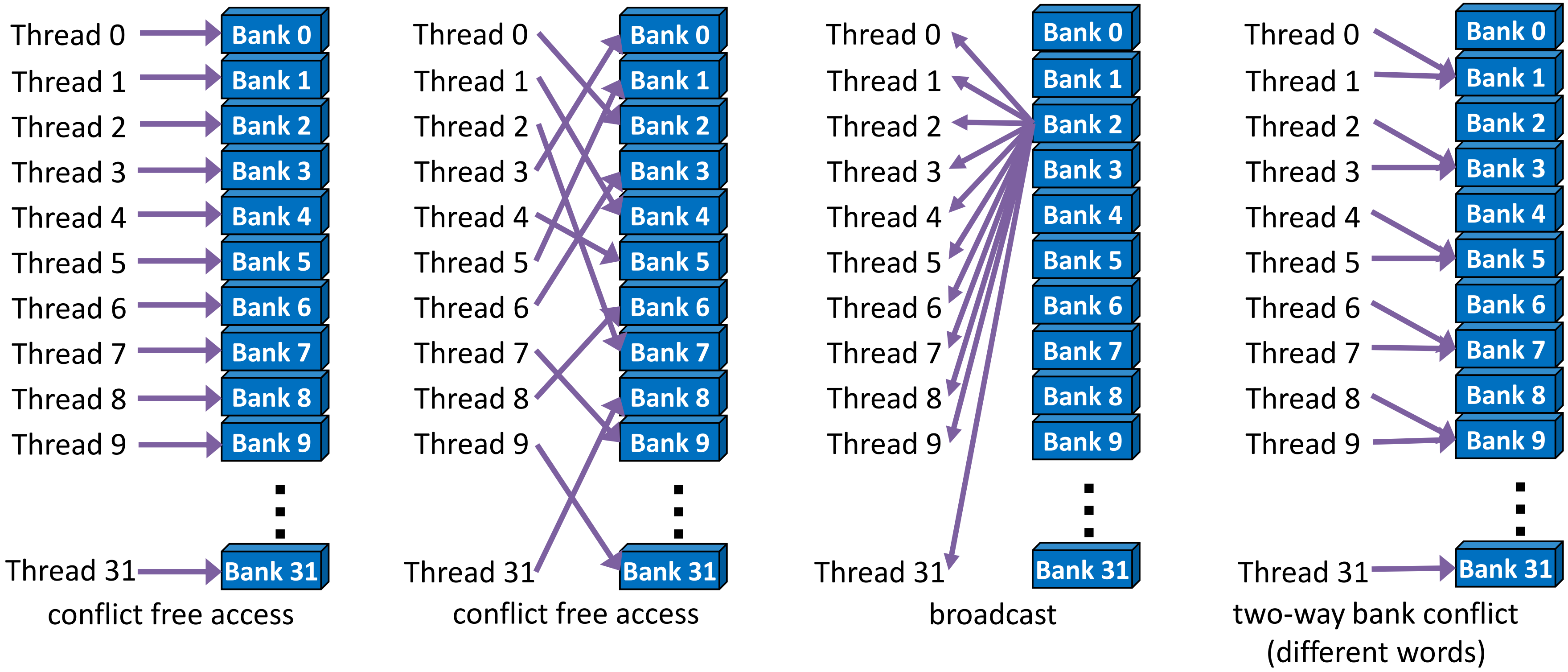
    bank = (address/8) % 32

    bank = (address/4) % 32

Shared memory cont'd

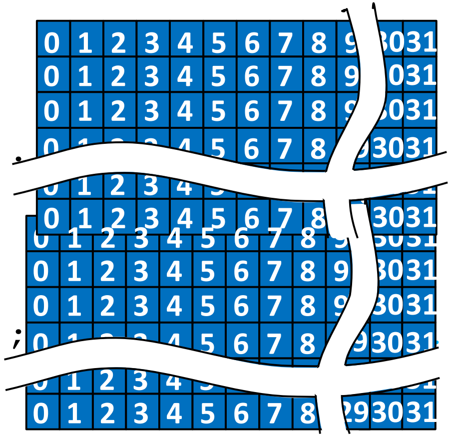
- Key to good performance is the access pattern
- Conflict free access: all threads within a warp access different banks (mind Kepler exceptions)
- Multicast: all threads accessing the same word are served with one transaction
- Serialization: multiple access conflicts will be serialized
- Introduce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

Shared memory cont’d



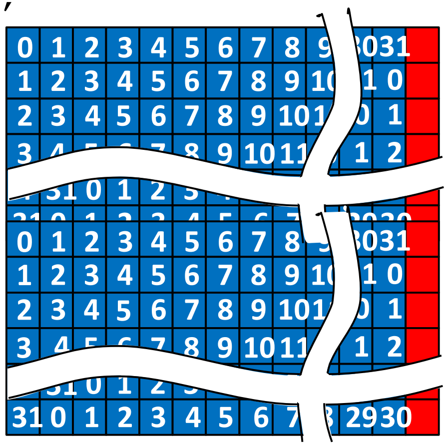
Shared memory cont'd

```
__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 cont'd

```
__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;   //for cc>=2 faster than if(threadId == 0)
  __syncthreads();
  while(run)
  {
    ...
    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 transp(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
- Indexed access

```
__global__ void kernel(...)
{
  uint mydata[8]; //will be spilled to local memory
  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
- Indexed access

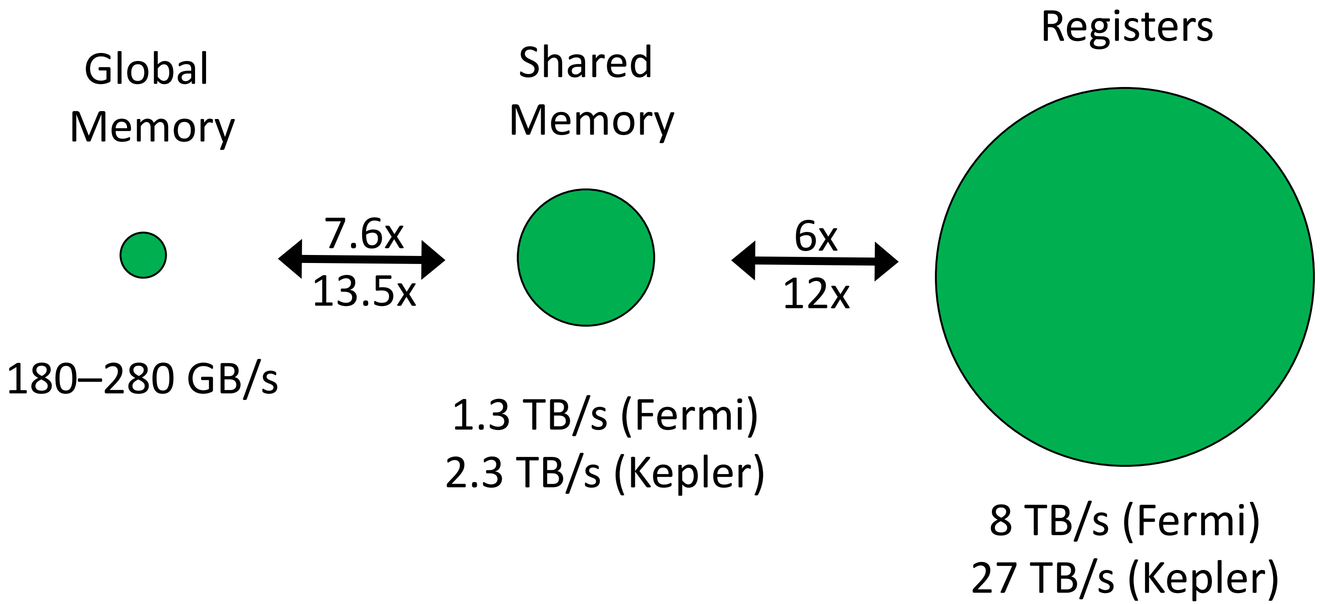
```
__global__ void kernel(...)
{
  __shared__ uint allmydata[8*BlockSize];
  uint *mydata = allmydata + 8*threadIdx.x;
  for(uint i = 0; i < 8; ++i)
    mydata[i] = complexfunc(i, threadIdx.x);
  uint res = 0;
  for(uint i = 0; i < 64; ++i)
   res += secondfunc(mydata[(threadIdx.x + i) % 8],
      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

Shared memory use cases

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



Questions