GPU Memory

Leiming Yu
yu.lei@husky.neu.edu

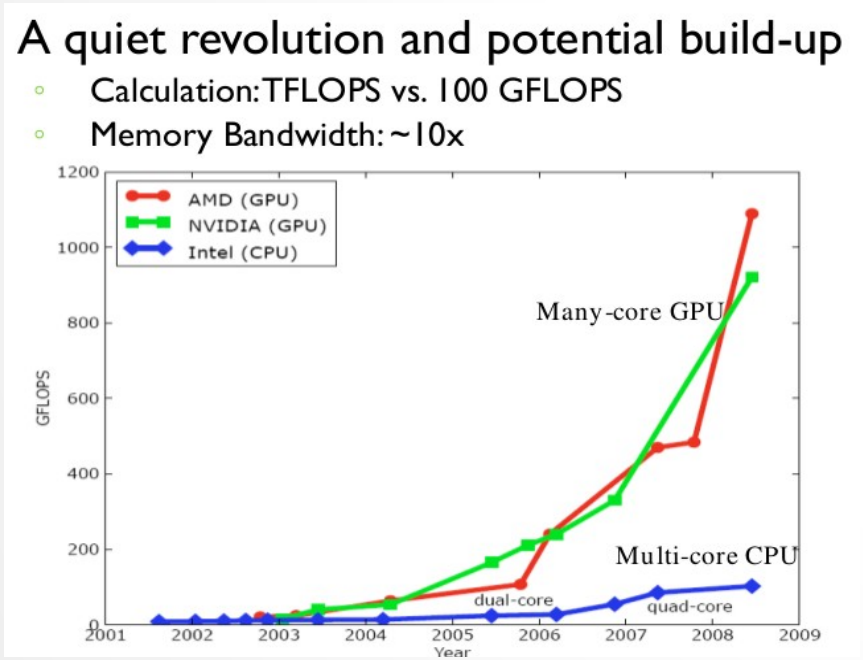
Topics

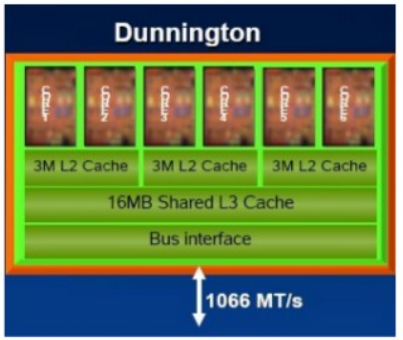
- Shared memory
- Matrix multiplication optimization
- Bank conflicts
- Global memory coalescing
- Other memory types
  - Constant
  - Texture
  - Read-only cache

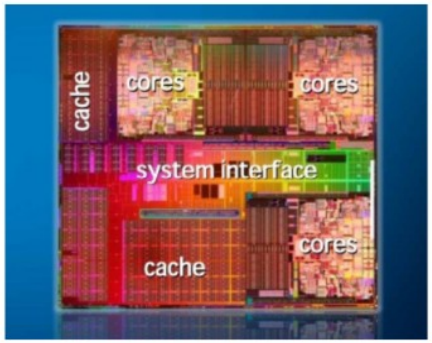
Catch up

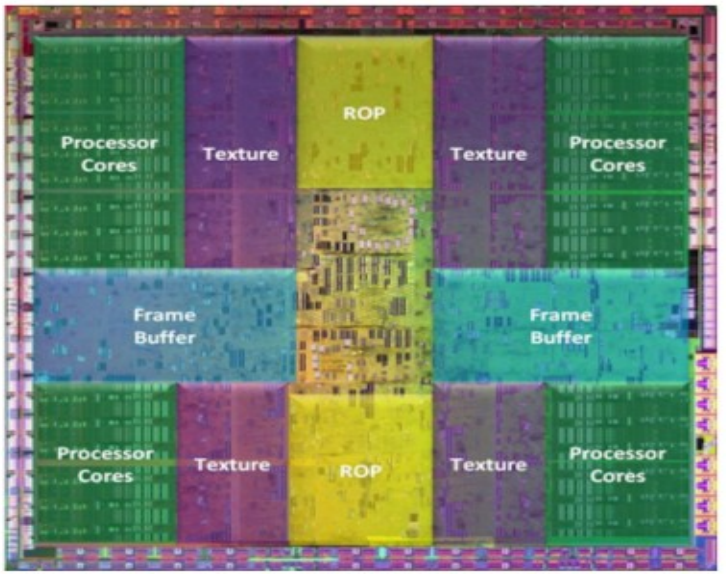
- High throughput and energy-efficient than cpu

- More parallel components









https://www.cvg.ethz.ch/teaching/2011spring/gpgpu/

Catch up

Flynn's Taxonomy :
a classification of computer architecture

S I S D **Single Instruction, Single Data**	**S I M D** **Single Instruction, Multiple Data**
M I S D **Multiple Instruction, Single Data**	**M I M D** **Multiple Instruction, Multiple Data**

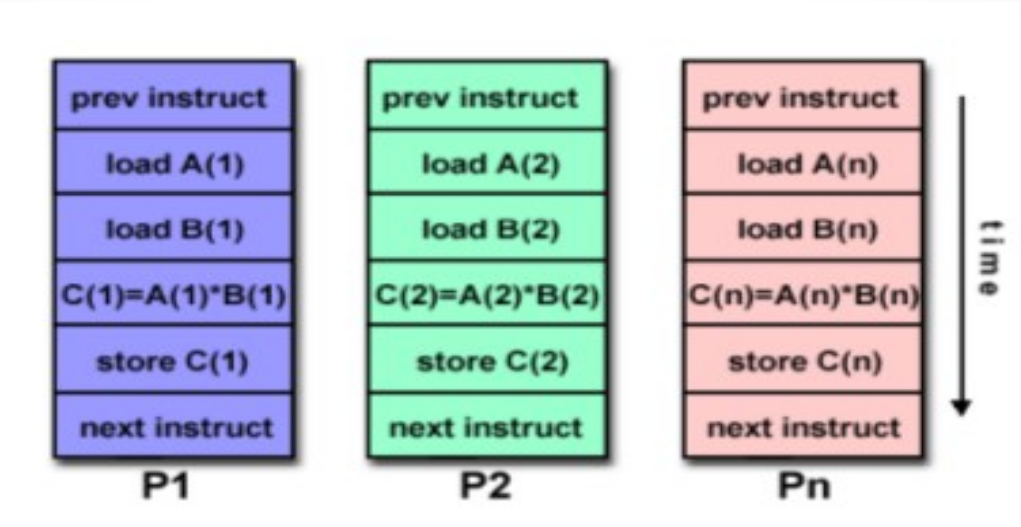
https://www.cvg.ethz.ch/teaching/2011spring/gpgpu/

Catch up

[1] SISD:

One instruction for one data stream per clock cycle

[2] SIMD:



[3] MISD:

One data stream for multiple processing units

[4] MIMD:

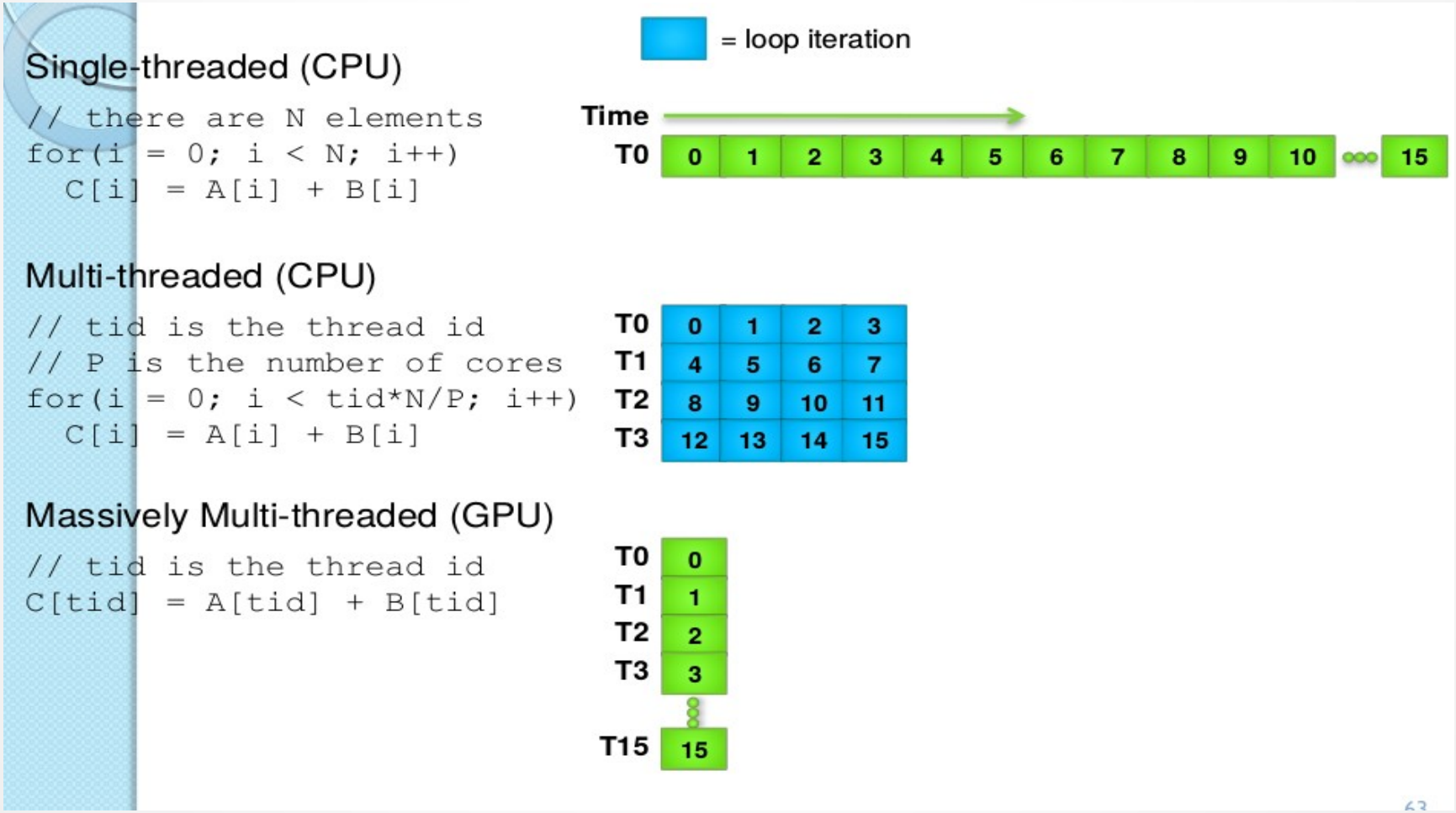
Processors work on different data streams,
Executing different data

https://www.cvg.ethz.ch/teaching/2011spring/gpgpu/

Catch up

GPU:

SPMD programming model

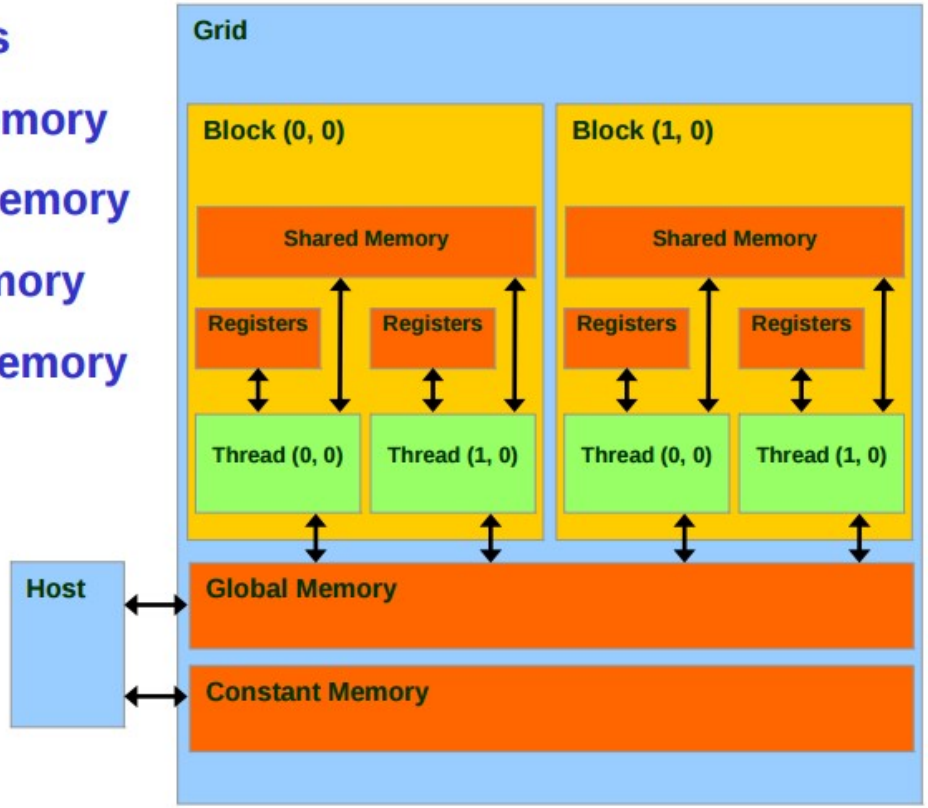


https://www.cvg.ethz.ch/teaching/2011spring/gpgpu/

GPU Memory Hierarchy

Each thread can:

- Read/write per-thread **registers**
- Read/write per-thread **local memory**
- Read/write per-block **shared memory**
- Read/write per-grid **global memory**
- Read/only per-grid **constant memory**



https://www.cvg.ethz.ch/teaching/2011spring/gpgpu/

GPU Memory Hierarchy

- Type Qualifier table

Variable declaration	Memory	Scope	Lifetime
`int LocalVar;`	register	thread	thread
`int LocalArray[10];`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- Notes:
  - `__device__` not required for `__local__`, `__shared__`, or `__constant__`
  - Automatic variables without any qualifier reside in a register
    - Except arrays that reside in local memory
    - Or not enough registers available for automatic variables

https://www.cvg.ethz.ch/teaching/2011spring/gpgpu/

GPU Memory Hierarchy

CUDA Type Qualifiers

- Type Qualifier table / performance

Variable declaration	Memory	Performance penalty
`int LocalVar;`	register	1x
`int LocalArray[10];`	local	100x
`__device__ __shared__ int SharedVar;`	shared	1x
`__device__ int GlobalVar;`	global	100x
`__device__ __constant__ int ConstantVar;`	constant	1x

- Notes (for G80, somewhat simplified)
  - Scalar vars reside in on-chip registers (fast)
  - Shared vars resides in on-chip memory (fast)
  - Local arrays and global variables reside in off-chip memory (slow)
  - Constants reside in cached off-chip memory

https://www.cvg.ethz.ch/teaching/2011spring/gpgpu/

Matrix Multiplication

* launch 2D grid
* compute element-wise output
  C[i][j] = sum(A[i][k] * B [k][j])

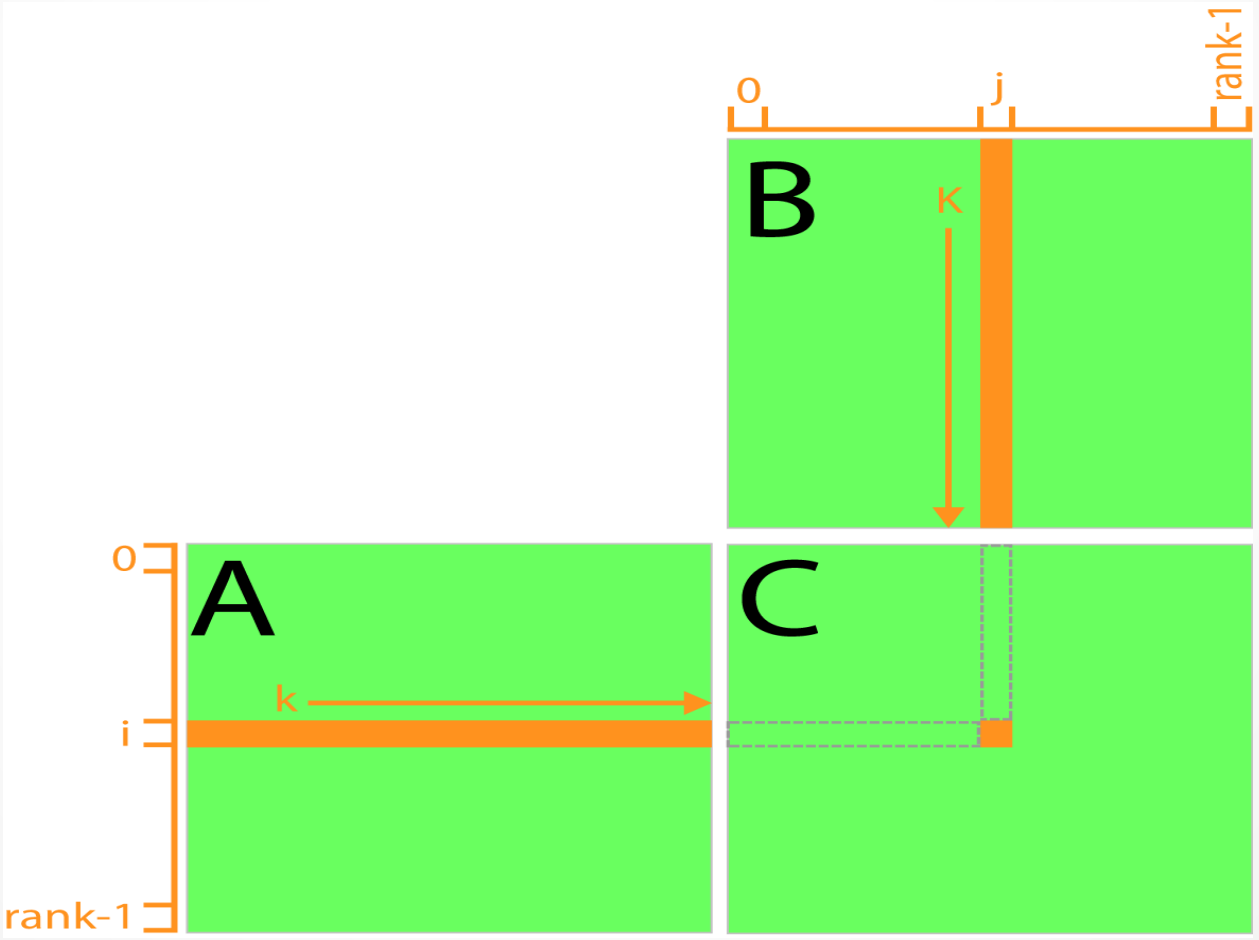
```
__global__ void MatrixMultiplyKernel_GlobalMem( float* C, const float* A, const float* B, unsigned int rank )
{
    // Compute the row index
    unsigned int i = ( blockDim.y * blockIdx.y ) + threadIdx.y;
    // Compute the column index
    unsigned int j = ( blockDim.x * blockIdx.x ) + threadIdx.x;

    unsigned int index = ( i * rank ) + j;
    float sum = 0.0f;
    for ( unsigned int k = 0; k < rank; ++k )
    {
        sum += A[i * rank + k] * B[k * rank + j];
    }
    C[index] = sum;
}
```

https://www.cvg.ethz.ch/teaching/2011spring/gpgpu/

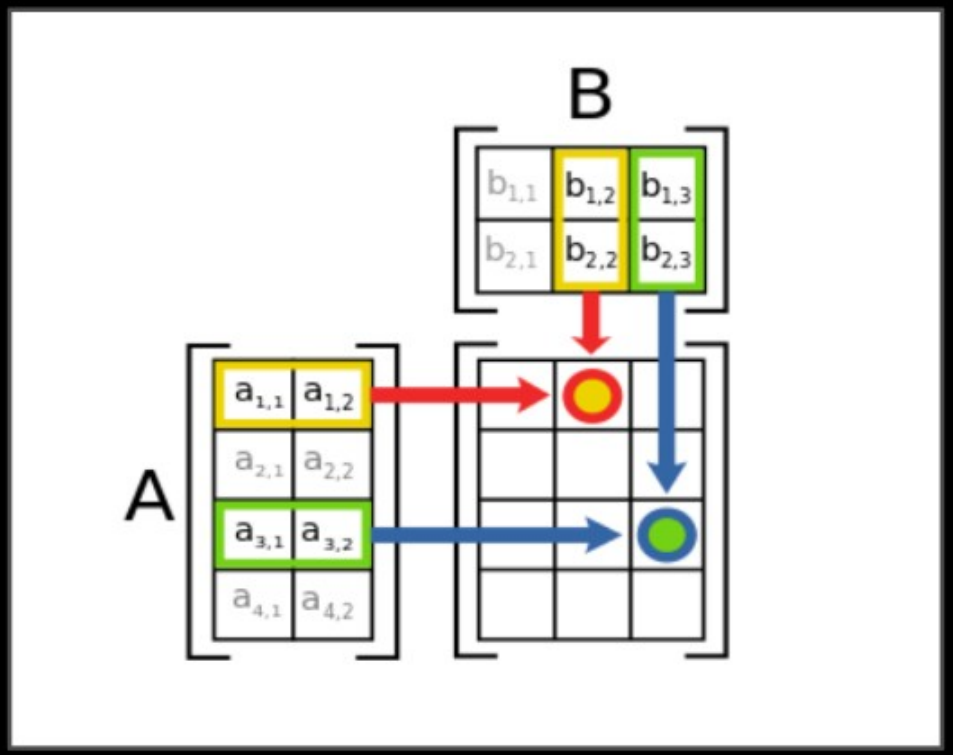
Matrix Multiplication

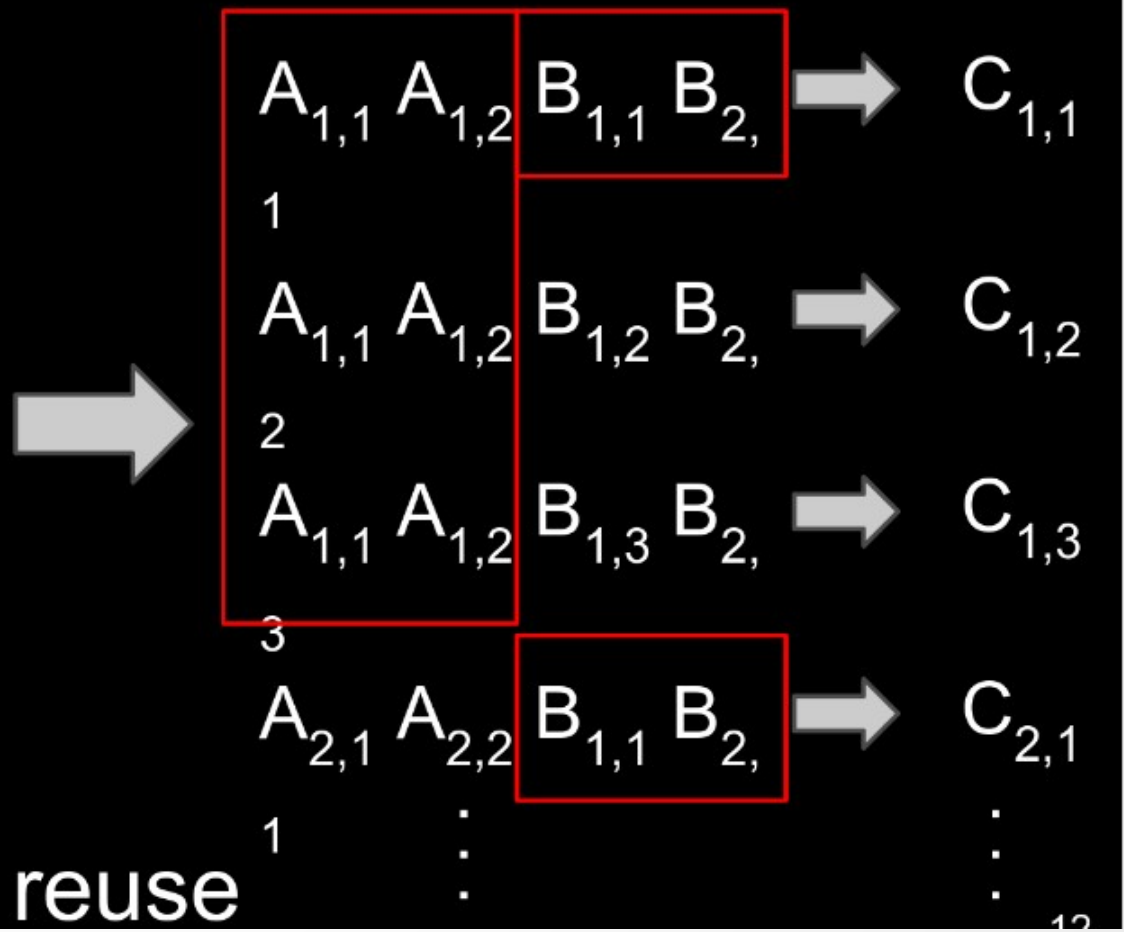
* launch 2D grid
* compute element-wise output
  C[i][j] = sum(A[i][k] * B [k][j])



Matrix Multiplication (shared mem.)

Problem with naive version

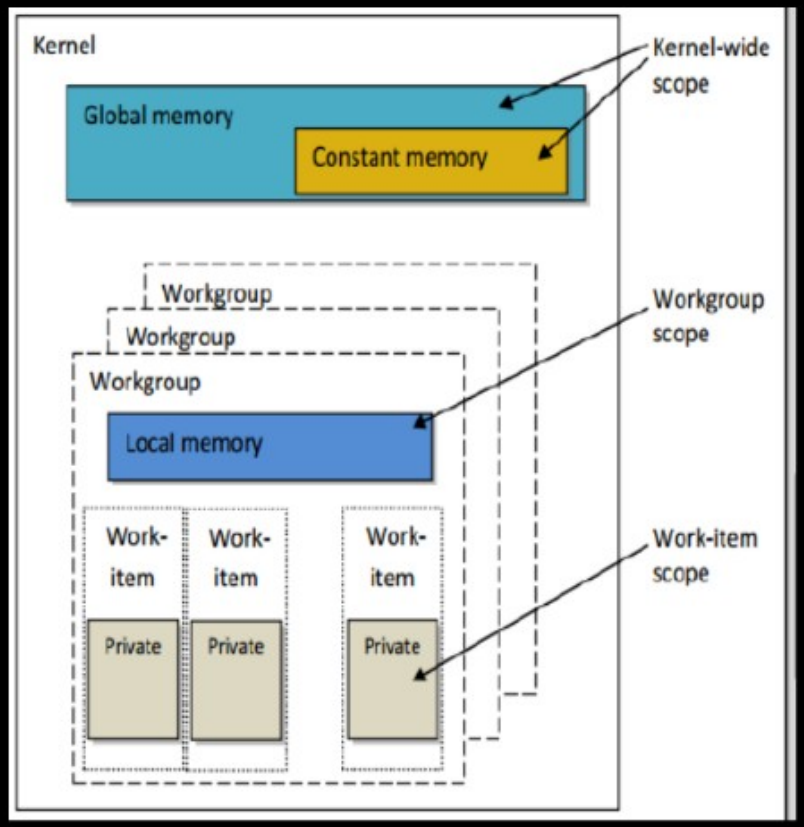




Data reuse

Matrix Multiplication (shared mem.)

Solution



- Local memory for data reuse
- Chunk the matrix into sub-matrices for parallel computation and good data locality

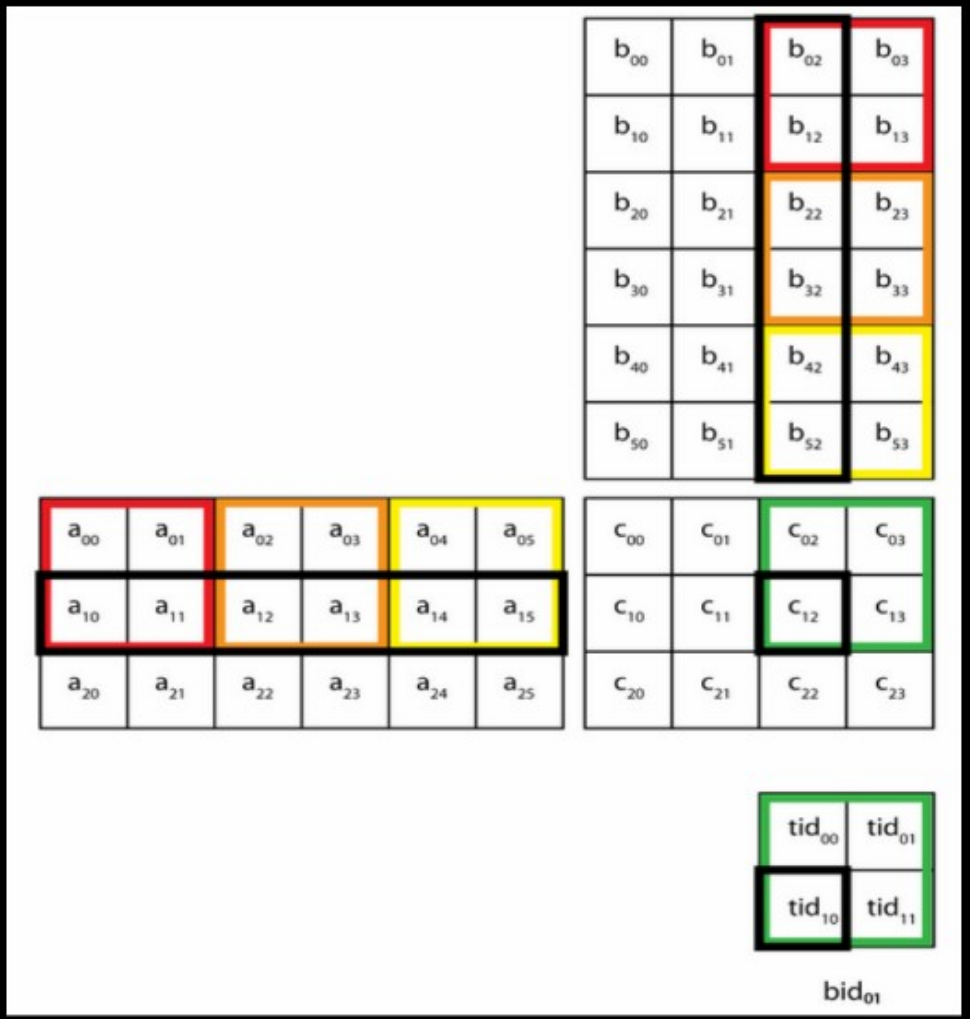
Shared Memory

Shared Memory/Cache

- On-chip memory
- Per SM: Combination of L1 cache and shared memory
  - 16KB + 48 KB shared memory + L1
  - 48KB + 16 KB shared memory + L1
- L2 cache shared by all SMs
- Cache accesses to local or global memory, including temporary register spills
- Cache inclusion (L1 $\subset$ L2?) partially configurable on per-access basis with mem. ref. instruction modifiers
- 128 byte cache line size

Matrix Multiplication (shared mem.)

Scheme



Matrix Multiplication (shared mem.)

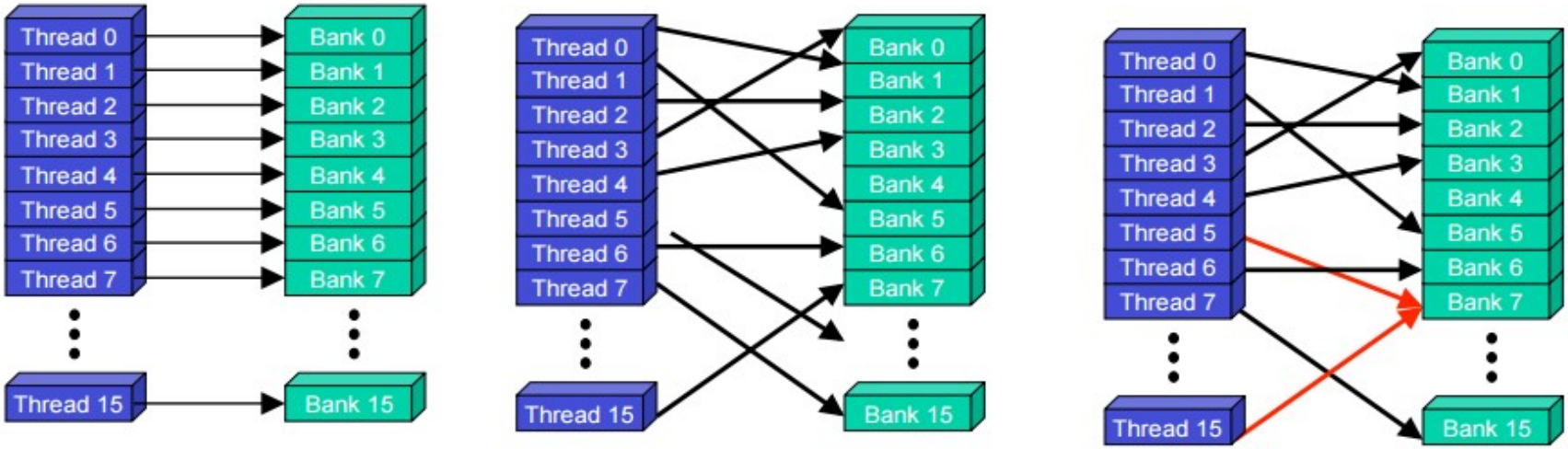
Congratulations!

You are leveled up ^~^

Bank Conflicts

Shared memory bank access

- Load/store of *n* addresses spanning *n* distinct memory banks can be serviced simultaneously, effective BW = $\times n$ a single bank's
- Each bank can service 1 address / cycle (bcast, too)
- Access to shared memory is fast unless…
  - 2 or more instructions in a 1/2 warp access different banks: we have a *conflict*
  - Exception: if all access the same bank: broadcast




```
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] = a[idx]+1.0f;
```

Bank Conflicts

- Successive 32-bit words assigned to successive banks
- For devices of compute capability 2.x [Fermi]
  - Number of banks = 32
  - Bandwidth is 32 bits per bank per **2** clock cycles
  - Shared memory request for a warp is not split
  - Increased susceptibility to conflicts
  - But no conflicts if access to bytes in same 32 bit word
  - Unlike 1.x, no bank conflicts here in the code example
- For devices of compute capability 1.x [Lilliput]
  - Number of banks = 16
  - Bandwidth is 32 bits per bank per clock cycle
  - Shared memory request for a warp is split in two
  - No conflict occurs if only one memory location per bank is accessed by a half warp of threads

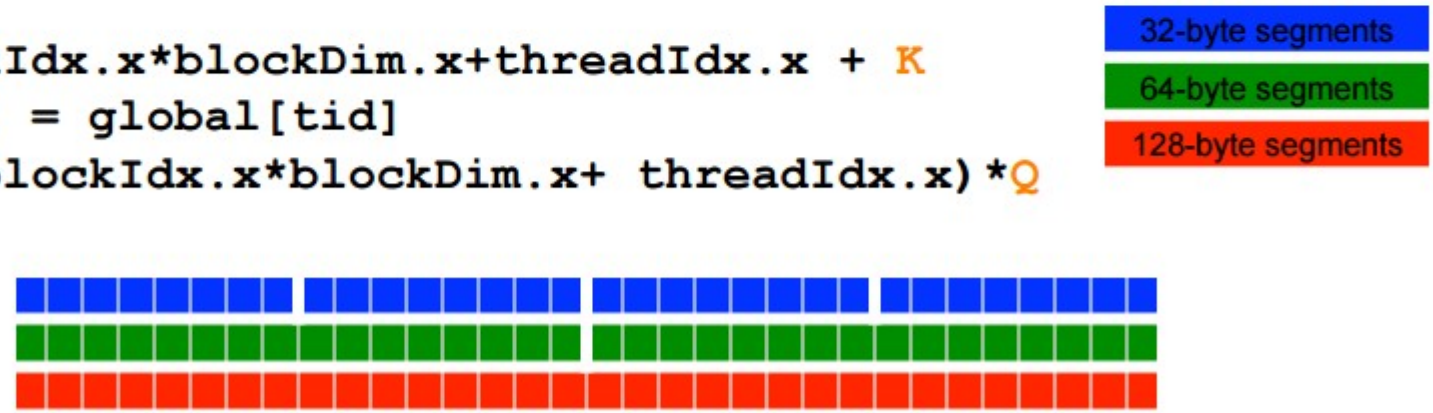
Global Memory

- If accessed word > 4 bytes, warp's memory request split into separate, independently issued 128-byte memory requests
- Non-atomic, concurrent writes within a warp: writer not defined

Global Memory

- Global memory accesses in units of 32, 64, 128 B
- Consecutive addresses read quickly (K=0; Q=1)
- Certain non-sequential access patterns to global memory degrade performance K mod 16 ≠ 0; Q≠1)
- Accesses organized by half warps (16 threads) can be done in one or two transactions, under certain conditions (32, 64 and 128 byte segments)

```
tid = blockIdx.x*blockDim.x+threadIdx.x + K
shared[tid] = global[tid]
int tid =(blockIdx.x*blockDim.x+ threadIdx.x)*Q
```



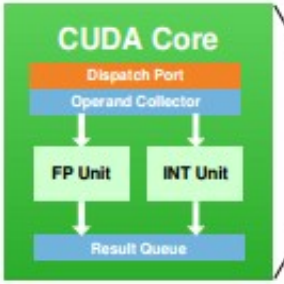
Nvidia Corp.

Global Memory

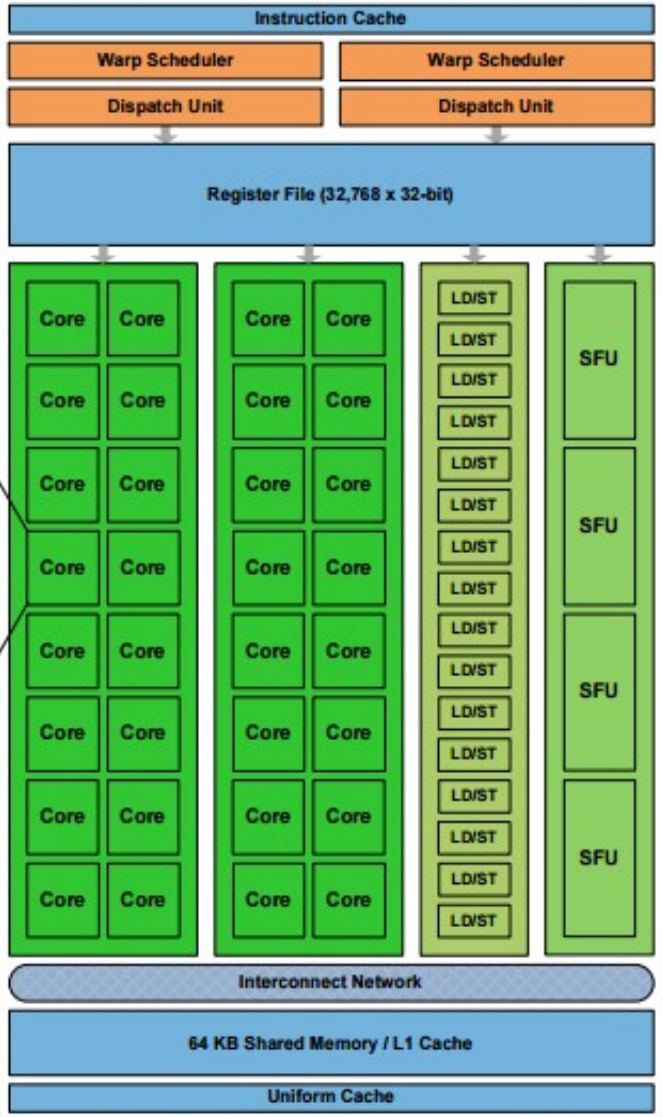
Third Generation Streaming Multiprocessor

The third generation SM introduces several architectural innovations that make it not only the most powerful SM yet built, but also the most programmable and efficient.

512 High Performance CUDA cores



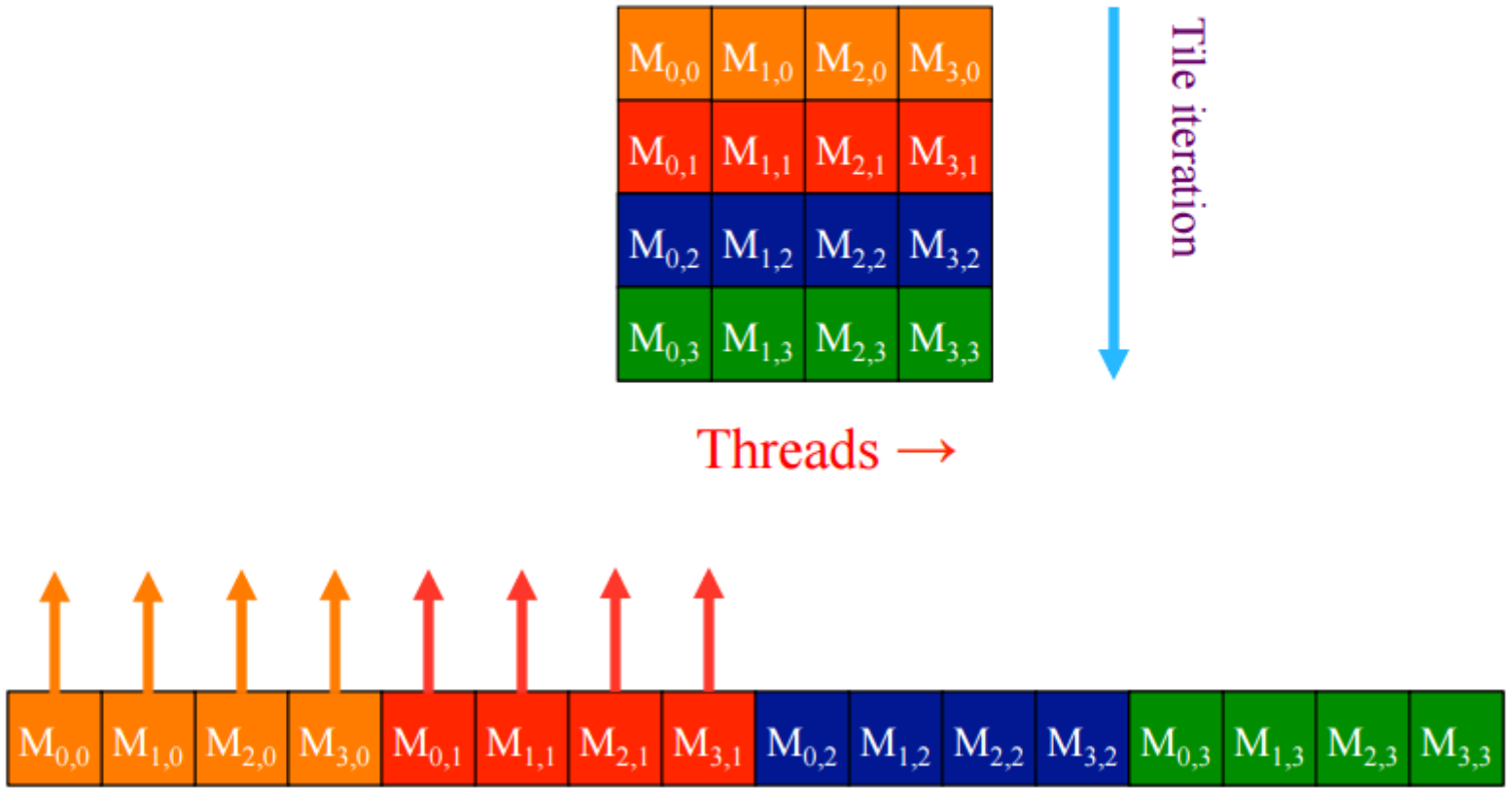
Each SM features 32 CUDA processors—a fourfold increase over prior SM designs. Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). Prior GPUs used IEEE 754-1985 floating point arithmetic. The Fermi architecture implements the new IEEE 754-2008 floating-point standard, providing the fused multiply-add (FMA) instruction for both single and double precision arithmetic. FMA improves over a multiply-add (MAD) instruction by doing the multiplication and addition with a single final rounding step, with no loss of precision in the addition. FMA is more accurate than performing the operations separately. GT200 implemented double precision FMA.



Fermi Streaming Multiprocessor (SM)

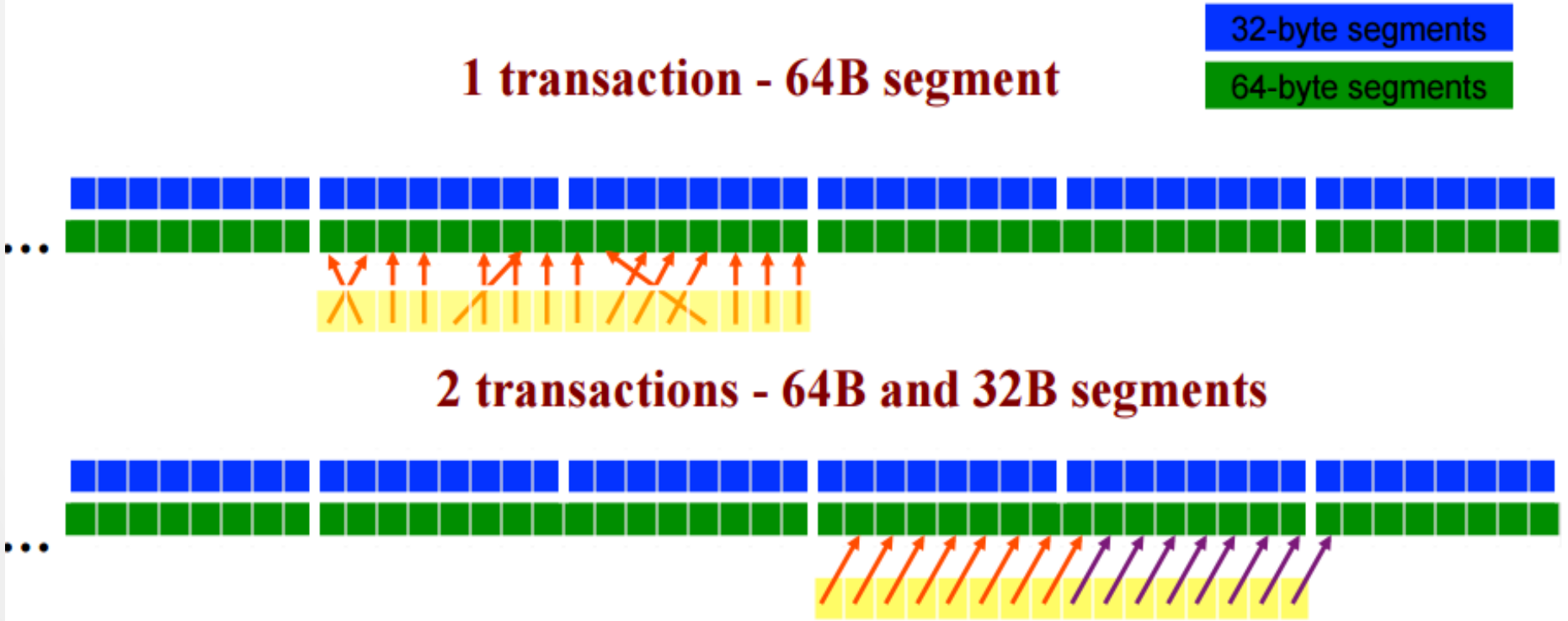
Global Memory

- Simplest: addresses are contiguous across threads
- Accesses organized by half warps (16 threads)



Memory coalescing (compute capability ≥1.2)

- Find the segment containing the address request of the lowest numbered active thread
- Find all other active threads requesting in same segment
- Reduce transaction size (if possible)
- Mark the serviced threads as inactive
- Repeat until all threads in ½ warp are complete

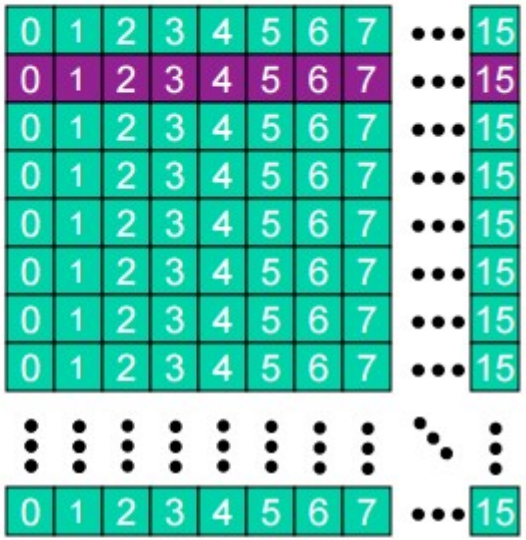


Nvidia Cuda C Programming Guide: **Appendix G.3.2.2**

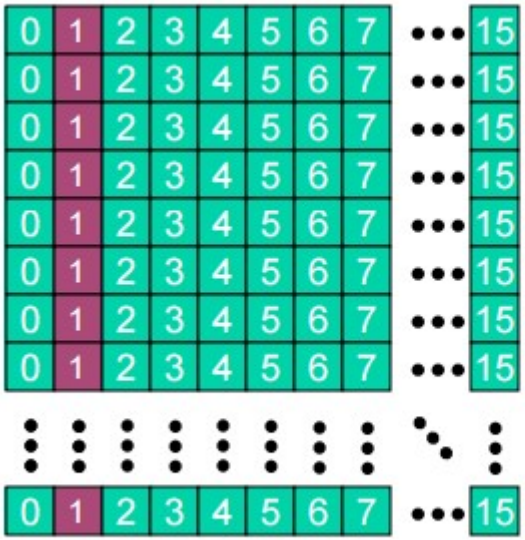
Coalescing with 2d arrays

- All warps in a block access consecutive elements within a row as they step through neighboring columns

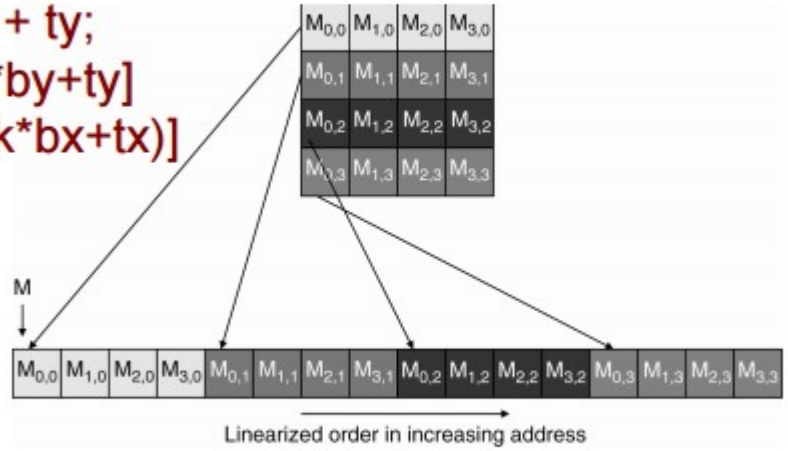
```
I =  blockIdx.y*by + ty;
J =  blockIdx.x*bx + tx;
int tx = threadIdx.x
a[ty][tx] = A[I*N+k*by+tx]
b[ty][tx] = B[J+N*(k*bx+ty)]
```



- Accesses by threads in a block along a column don't coalesce




```
I =  blockIdx.x*bx + tx;
J =  blockIdx.y*by + ty;
a[tx][ty] = A[I*N+k*by+ty]
b[ty][tx] = B[J+N*(k*bx+tx)]
```

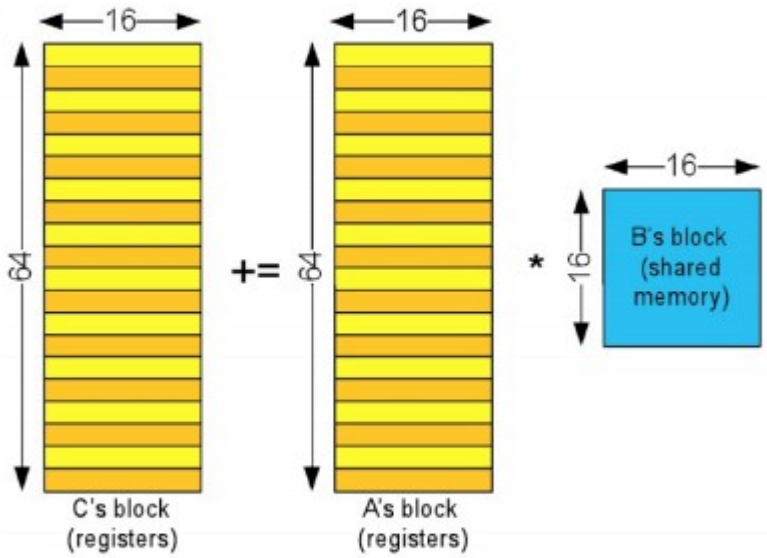


Volkov and Demmel's SGEMM

```
Vector length: 64 //stripmined into two warps by GPU
Registers: a, c[1:16] //each is 64-element vector
Shared memory: b[16][16] //may include padding

Compute pointers in A, B and C using thread ID
c[1:16] = 0
do
   b[1:16][1:16] = next 16×16 block in B or B^T
   local barrier //wait until b[][] is written by all warps
   unroll for i = 1 to 16 do
      a = next 64×1 column of A
      c[1] += a*b[i][1]     // rank-1 update of C's block
      c[2] += a*b[i][2]     // data parallelism = 1024
      c[3] += a*b[i][3]     // stripmined in software
      ...                   // into 16 operations
      c[16] += a*b[i][16]   // access to b[][] is stride-1
   endfor
   local barrier //wait until done using b[][]
   update pointers in A and B
repeat until pointer in B is out of range
Merge c[1:16] with 64×16 block of C in memory
```

Figure 4: The structure of our matrix-matrix multiply routines.



SGEMM Code

```
__global__ void sgemmNN( const float *A, int lda, const float *B, int ldb, float* C, int ldc, int k, float alpha, float beta )
{
    A += blockIdx.x * 64 + threadIdx.x + threadIdx.y*16;
    B += threadIdx.x + ( blockIdx.y * 16 + threadIdx.y ) * ldb;
    C += blockIdx.x * 64 + threadIdx.x + (threadIdx.y + blockIdx.y * ldc ) * 16;
    __shared__ float bs[16][17];
    float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    const float *Blast = B + k;
    do
    {
#pragma unroll
        for( int i = 0; i < 16; i += 4 )
            bs[threadIdx.x][threadIdx.y+i]  = B[i*ldb];
        B += 16;
        __syncthreads();
#pragma unroll
        for( int i = 0; i < 16; i++, A += lda )
        {
            c[0] += A[0]*bs[i][0];    c[1] += A[0]*bs[i][1];    c[2] += A[0]*bs[i][2];    c[3] += A[0]*bs[i][3];
            c[4] += A[0]*bs[i][4];    c[5] += A[0]*bs[i][5];    c[6] += A[0]*bs[i][6];    c[7] += A[0]*bs[i][7];
            c[8] += A[0]*bs[i][8];    c[9] += A[0]*bs[i][9];    c[10] += A[0]*bs[i][10];c[11] += A[0]*bs[i][11];
            c[12] += A[0]*bs[i][12];c[13] += A[0]*bs[i][13];c[14] += A[0]*bs[i][14];c[15] += A[0]*bs[i][15];
        }
        __syncthreads();
    } while( B < Blast );
    for( int i = 0; i < 16; i++, C += ldc )
        C[0] = alpha*c[i] + beta*C[0];
}
```

Compute pointers to the data

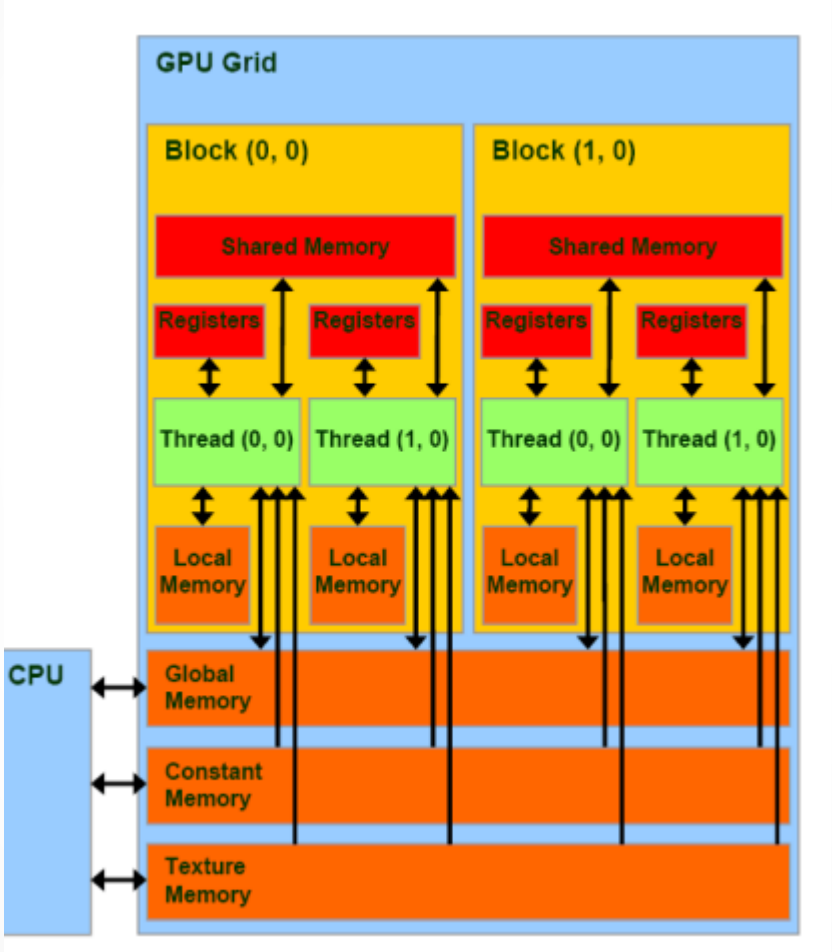
Declare the on-chip storage

Read next B's block

The bottleneck:
Read A's columns
Do Rank-1 updates

Store C's block to memory

Constant Memory



Constant Memory

Constant Memory II

- The mechanism for declaring memory constant is similar to the one we used of declaring a buffer as shared memory.
- The instruction to define constant memory is _constant_
- It must be declare out of the main body and the kernel.
- The instruction *cudaMemcpyToSymbol* must be used in order to copy the values to the kernel.

```
// CUDA global constants
  __constant__ int M;

int main(void)
{
...
  cudaMemcpyToSymbol("M", &M, sizeof(M));
...
}
```

Constant Memory

Constant Memory III

- The variables in constant memory are not necessary to be declared in the kernel invocation.

```
__global__ void kernel(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N)
  {
    a[idx] =  a[idx] *M;
  }
}
```

Constant Memory

```
int main(void)
{
  int M = 100;
  float *a_h, *a_d;  // Pointer to host & device arrays
  const int N = 4*4;  // Number of elements in arrays
  size_t size = N * sizeof(float);

  a_h = (float *)malloc(size);        // Allocate array on host
  cudaMalloc((void **) &a_d, size);   // Allocate array on device

  // Initialize host array and copy it to CUDA device
  for (int i=0; i<N; i++) a_h[i] = (float)i;

  printf("antes\n");
  for (int i=0; i<N; i++) printf("%d %f \n", i, a_h[i] );
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol("M", &M, sizeof(M));
  // Do calculation on device:
  int block_size = 4;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);

  kernel <<< n_blocks, block_size >>> (a_d, N);

  cudaMemcpy(a_h, a_d, N*sizeof(float), cudaMemcpyDeviceToHost);

  // Print results
  printf("\n despues \n");
  for (int i=0; i<N; i++) printf("%d %f \n", i, a_h[i] );

  // Cleanup
  free(a_h); cudaFree(a_d);
}
```

Constant Memory

Performance Considerations I

- Declaring memory as constant constrains our usage to be read-only. In taking on this constraint, we expect to get something in return.
- Reading from constant memory can conserve memory bandwidth when compared to reading the same data from global memory.
- There are two reasons why reading from the 64 KB of constant memory can save bandwidth over standard reads of global memory:
  - A single read from constant memory can be broadcast to other read by threads, effectively saving up to 15 reads.
  - Constant memory is cached, so consecutive read of the same address will not incur any additional memory traffic.

Warp

In the CUDA architecture, a warp refers to a collection of 32 thread that are "woven together" and get executed in locksptep.

Constant Memory

Performance Considerations II

- NVIDIA hardware can broadcast a single memory read to each half-warp.
- If every thread in a half-warp requests data from the same address in constant memory, your GPU will generate only a single read request and subsequently broadcast the data to every thread.
- If you are reading a lot of data from constant memory, you will generate on 1/16 (6%) of the memory traffic as you would when using global memory.

Constant Memory

Performance Considerations III

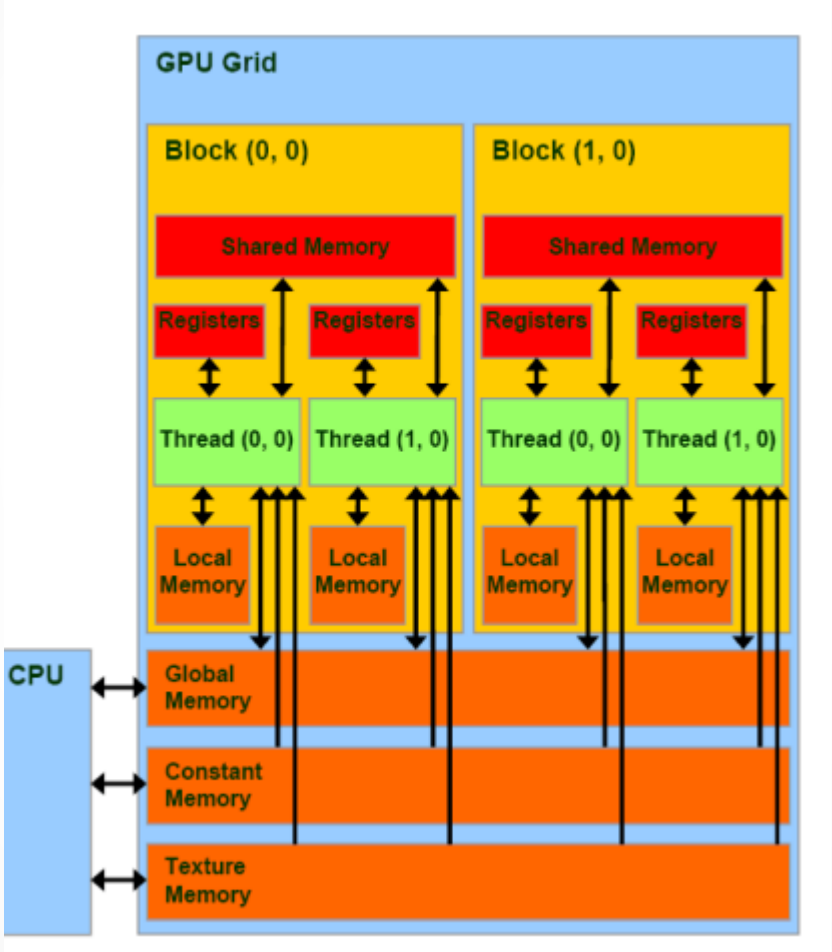
- The savings don't stop at at 94% reduction in bandwidth when reading constant memory.
- Furthermore, the hardware can cache the constant data on the GPU.
- So after the first read from an address in constant memory, other half-warps requesting the same address, and therefore, hitting the constant cache, will generate not additional memory traffic.
- After caching the data, every other thread avoids generating memory traffic as a result of one of the two constant memory benefits:
  - It receives the data in a half-warp broadcast.
  - It retrieves the data from the constant memory cache.

Constant Memory

Performance Considerations IV

- Unfortunately, there can be a downgrade of the performance when using constant memory.
- However, the half-warp broadcast feature can degrade the performance when all 16 threads read different addresses.
- If all 16 threads in a half-warp need different data from constant memory, the 16 different reads get serialized.

Texture Memory



Texture Memory

* **Texture References**
* Texture Objects

```
#define N 1024
texture<float, 1, cudaReadModeElementType> tex;

// texture reference name must be known at compile time
__global__ void kernel() {
  int i = blockIdx.x *blockDim.x + threadIdx.x;
  float x = tex1Dfetch(tex, i);
  // do some work using x...
}

void call_kernel(float *buffer) {
  // bind texture to buffer
  cudaBindTexture(0, tex, buffer, N*sizeof(float));

  dim3 block(128,1,1);
  dim3 grid(N/block.x,1,1);
  kernel <<<grid, block>>>();

  // unbind texture from buffer
  cudaUnbindTexture(tex);
}

int main() {
  // declare and allocate memory
  float *buffer;
  cudaMalloc(&buffer, N*sizeof(float));
```

Texture Memory

* Texture References
* **Texture Objects**

```
// declare and allocate memory
float *buffer;
cudaMalloc(&buffer, N*sizeof(float));

// create texture object
cudaResourceDesc resDesc;
memset(&resDesc, 0, sizeof(resDesc));
resDesc.resType = cudaResourceTypeLinear;
resDesc.res.linear.devPtr = buffer;
resDesc.res.linear.desc.f = cudaChannelFormatKindFloat;
resDesc.res.linear.desc.x = 32; // bits per channel
resDesc.res.linear.sizeInBytes = N*sizeof(float);

cudaTextureDesc texDesc;
memset(&texDesc, 0, sizeof(texDesc));
texDesc.readMode = cudaReadModeElementType;

// create texture object: we only have to do this once!
cudaTextureObject_t tex=0;
cudaCreateTextureObject(&tex, &resDesc, &texDesc, NULL);

call_kernel(tex); // pass texture as argument

// destroy texture object
cudaDestroyTextureObject(tex);
```

Texture Memory

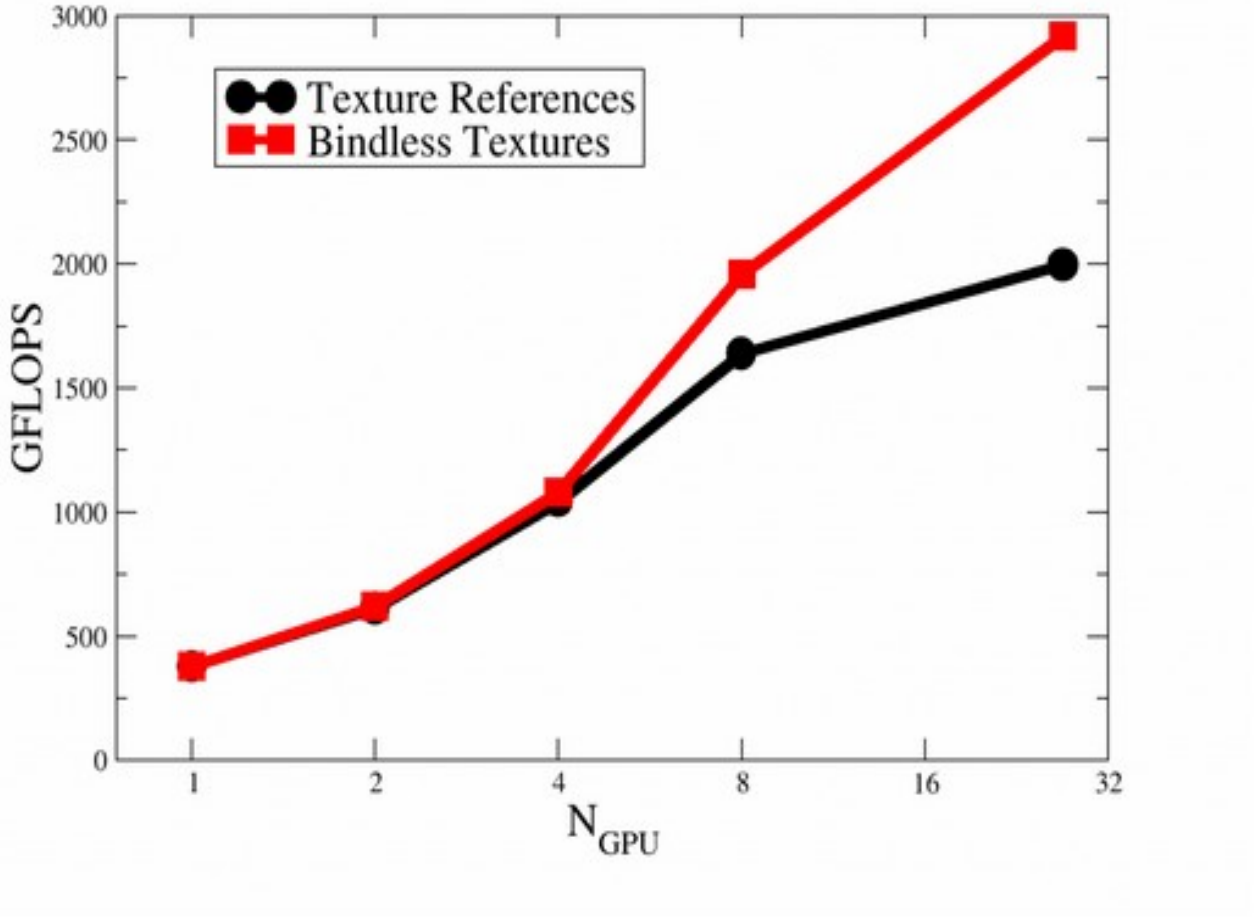
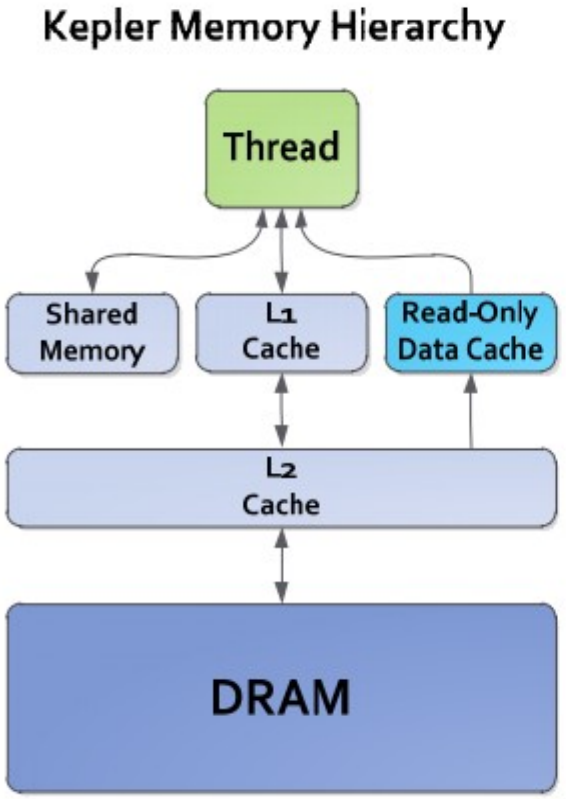


Figure 1: Strong-scaling performance of the QUDA solver using both texture references and texture objects.

Read-only Cache

Kepler Memory Subsystem – L1, L2, ECC

Kepler's memory hierarchy is organized similarly to Fermi. The Kepler architecture supports a unified memory request path for loads and stores, with an L1 cache per SMX multiprocessor. Kepler GK110 also enables compiler-directed use of an additional new cache for read-only data, as described below.



Homework 3

1. Write optimized matrix multiplication using shared memory.

2. Profiling the performance for naïve and opt version using nvvp/nvprof. Plot the figure where the performance counters can show your optimizations. For example, how the global memory bandwidth utilization changes.