Introduction to GPU Programming

Volodymyr (Vlad) Kindratenko

Innovative Systems Laboratory @ NCSA

Institute for Advanced Computing Applications and Technologies (IACAT)

Part III

- Performance considerations
  - Host side
    - Events, streams, compute capability
    - Host memory, data transfer
    - Thread management
  - Device side
    - Global memory, memory coalescing
    - Shared memory, registers
    - Threads, blocks, occupancy
    - Arithmetic instructions, control flow
  - Final recommendations

- Hands-on: optimizing matrix multiplication

- Break

Events

- Events can be asynchronously inserted and then recorded when all tasks preceding the event have completed

```
cudaEvent_t start, stop;
float time;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);
kernel<<<grid, threads>>> ( d_odata, d_idata, size_x, size_y, NUM_REPS);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);  // in milliseconds

cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Streams

- Streams are used to manage concurrency
- Stream is a sequence of commands that execute in order
  - Created as a stream object
  - Used in kernel launch and memory copy operations

```
cudaStream_t s;
cudaStreamCreate(&s);
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, s);
kernel<<<grid, block, 0, s>>>(otherData_d);
```

Compute Capability

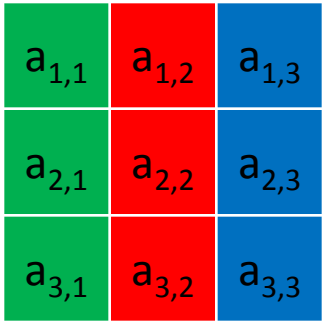
- Specs and supported features of a given GPU depend on its compute capability
  - 1.0, 1.1, 1.2, 1.3, 2.0
- Before using a feature, it is a good idea to query the device at run-time to verify that the required feature is supported

```
cudaDeviceProp props;
cudaGetDeviceProperties(&props, device);
```

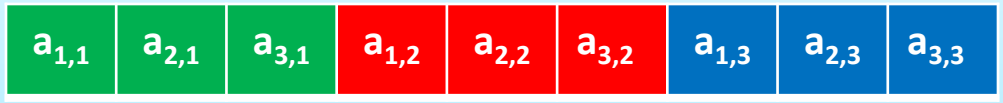
cudaDeviceProp struct

- int canMapHostMemory - Device can map host memory with cudaHostAlloc/cudaHostGetDevicePointer
- int clockRate - Clock frequency in kilohertz
- int computeMode - Compute mode
- int deviceOverlap - Device can concurrently copy memory and execute a kernel
- int integrated - Device is integrated as opposed to discrete
- int kernelExecTimeoutEnabled - Specified whether there is a run time limit on kernels
- int major - Major compute capability
- int maxGridSize [3] - Maximum size of each dimension of a grid
- int maxThreadsDim [3] - Maximum size of each dimension of a block
- int maxThreadsPerBlock - Maximum number of threads per block
- size_t memPitch - Maximum pitch in bytes allowed by memory copies
- int minor - Minor compute capability
- int multiProcessorCount - Number of multiprocessors on device
- char name [256] - ASCII string identifying device
- int regsPerBlock - 32-bit registers available per block
- size_t sharedMemPerBlock - Shared memory available per block in bytes
- size_t textureAlignment - Alignment requirement for textures
- size_t totalConstMem - Constant memory available on device in bytes
- size_t totalGlobalMem - Global memory available on device in bytes
- int warpSize - Warp size in threads
- … … …

Memory Alignment

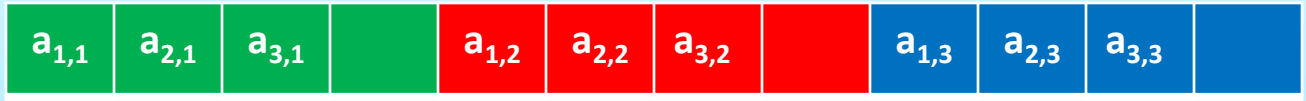


cudaMalloc(&dev_a, m*n*sizeof(float));



Matrix columns are not aligned at 64-bit boundary

cudaMallocPitch(&dev_a, &n, n*sizeof(float), m);



Matrix columns are aligned at 64-bit boundary

n is the allocated (aligned) size for the first dimension (the *pitch*), given the requested sizes of the two dimensions.

Memory Alignment Example

```
cudaMallocPitch((void**)&devPtr, &pitch, width * sizeof(float), height);

myKernel<<<100, 192>>>(devPtr, pitch);

// device code
__global__ void myKernel(float* devPtr, int pitch)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

Page-locked host memory

- Page-locked (or pinned) memory can be allocated on the host using cudaMallocHost() or cudaHostAlloc() subroutines
  - Higher PCIe transfer rate can be attained
  - By default memory block can be used only by the CPU thread that created it, but it also can be shared between any CPU threads when declared with cudaHostAllocPortable flag
  - By default pinned memory is cacheable, but it also can be allocated as write-combining by passing flag cudaHostAllocWriteCombined
    - Does not use L1/L2 CPU cache and is not snooped during PCIe data transfer = higher PCIe transfer bandwidth
    - Very slow when reading from it on the host, thus should only be used for writing on the host

Asynchronous data transfers

- cudaMemcpy() calls are blocking
- cudaMemcpyAsync() calls are non-blocking
  - Can be used to overlap computation on the host and data transfer and computation on the GPU

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
cpuFunction();
```

Asynchronous data transfers

- When using different streams, it is also possible to overlap data transfer with the kernel computation

```
cudaStreamCreate(&s1);
cudaStreamCreate(&s2);
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, s1);
kernel<<<grid, block, 0, s2>>>(otherData_d);
```

- Useful for double-buffering

Mapped host memory (zero copy)

- A block of page-locked host memory can also be mapped into the address space of the device by passing cudaHostAllocMapped flag

```
float *a_host, *a_device;
…
cudaGetDeviceProperties(&prop, 0);
if (!prop.canMapHostMemory) exit(0);
cudaSetDeviceFlags(cudaDeviceMapHost);

cudaHostAlloc((void **)&a_host, nBytes, cudaHostAllocMapped);
cudaHostGetDevicePointer((void **)&a_device, (void *)a_host, 0);
kernel<<<gridSize, blockSize>>>(a_device);
```

Concurrent kernel execution

- Some devices with compute capability 2.0 can execute multiple kernels simultaneously
  - Check for concurentKernel property before using this
- Max number of simultaneous kernels is currently 4

Thread management on the host

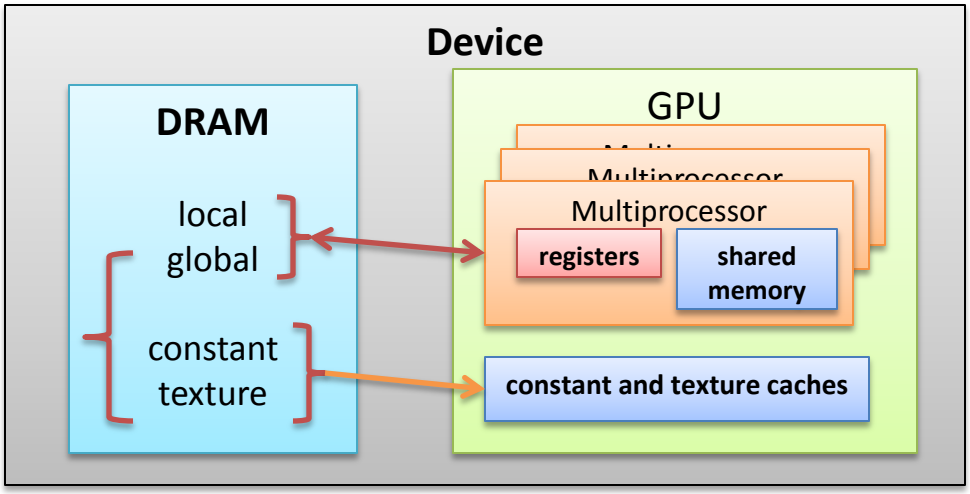
- cudaError_t cudaThreadExit (void)
  - Exit and clean up from CUDA launches

- cudaError_t cudaThreadSynchronize (void)
  - Wait for compute device to finish

```
kernel<<< dimGrid, dimBlock>>>( d_b, d_a );
cudaThreadSynchronize();
```

GPU memory spaces

- Global memory
  - Latency is on the order of several hundred cycles
- On-chip memory
  - 2 orders of magnitude lower latency than global memory
  - Order of magnitude higher bandwidth than global memory



Device memory bandwidth

- Theoretical bandwidth
  - GTX280 example
    - Double data rate (DDR)
    - RAM frequency: 1,107 MHz
    - Memory interface: 512 bits
    - $(1{,}107 \times 10^6 \times (512/8) \times 2) / 10^9 = 141.6$ GB/s
- Effective bandwidth
  - (bytes read + bytes written) / 10^9 / time

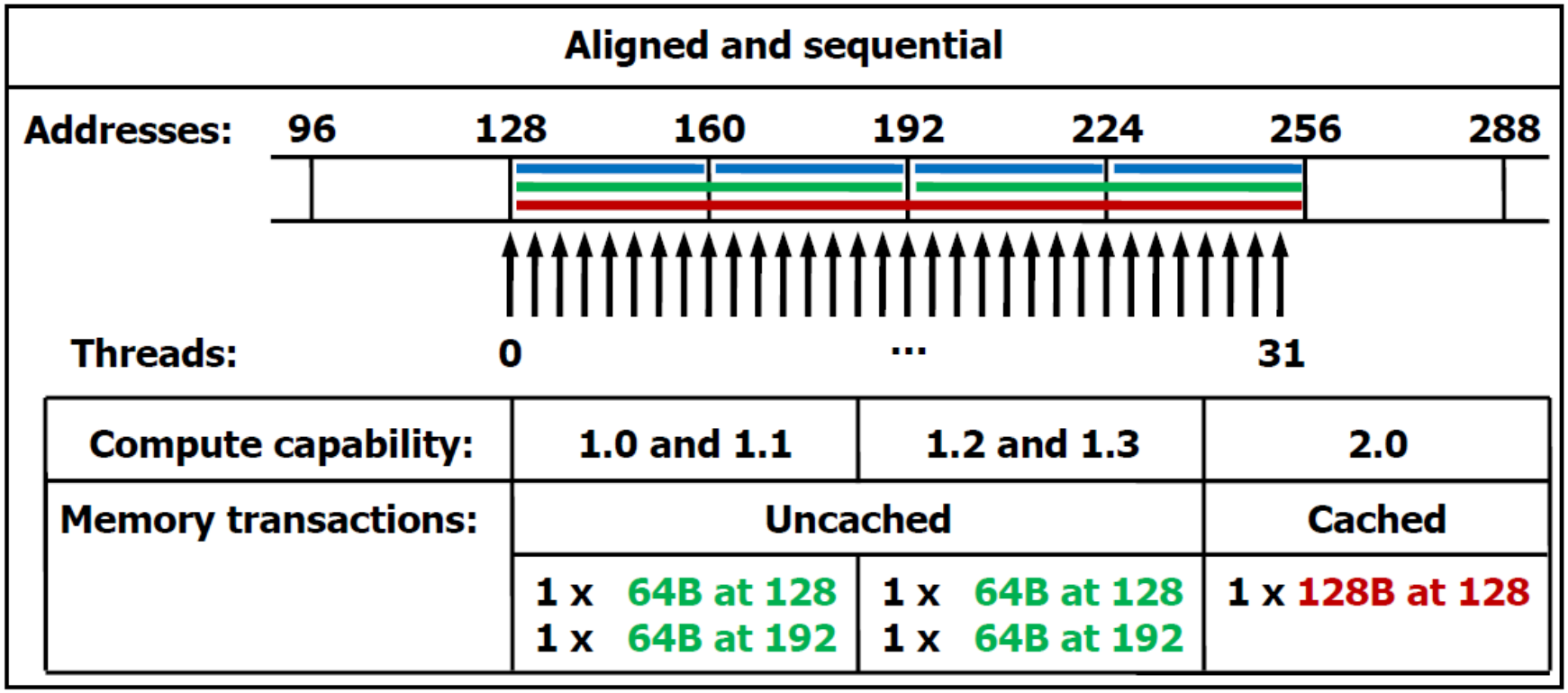
Global memory access

- Global memory resides in device memory
- Device memory is accessed via 32-, 64-, or 128-byte memory transactions
- These memory transactions must be naturally aligned
  - 32-, 64-, or 128-byte data segments should be aligned to the memory address which is a multiple of the corresponding size
- Global memory instructions support read/write word size of 1, 2, 4, 8, or 16 bytes
  - If size and alignment requirements are not fulfilled, multiple memory assess instructions will be generated
  - For structures, the size alignment requirements can be enforced by the compiler using the alignment specifiers __align__(8) or (16)
    - struct __align__(8) { float x, y };
    - struct __align__(16) { float x, y, z };

Coalesced access to global memory

- When a thread executes a global memory access instruction, memory accesses are coalesced for multiple threads into 32-, 64-, or 128-byte memory transactions
  - On devices with compute capability 1.x, global memory requests from a group of 16 threads (half-warp) are coalesced
  - On devices with compute capability 2.0, global memory requests from a group of 32 threads (warp) are coalesced
  - Threads must access the words in memory in sequence, e.g., k^{th} thread in a group of 16 threads must access k^{th} word
    - The size of the words accessed by the threads must be 4, 8, or 16 bytes
  - On devices with compute capability 2.0, global memory accesses are cached.
    - Each line in L1 or L2 caches is 128 bytes and maps to a 128-byte aligned segment in device memory

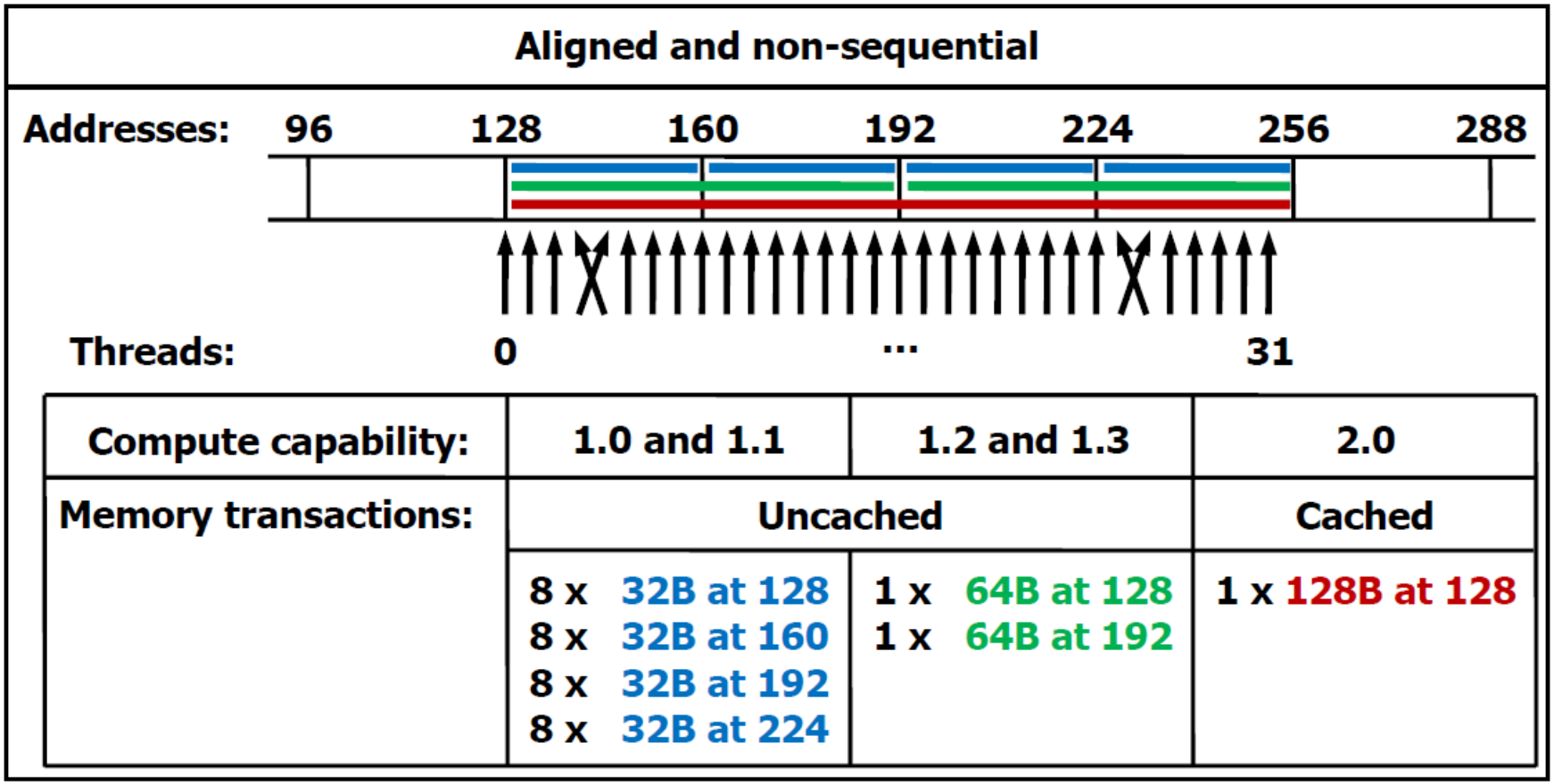
4-byte word per thread example



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Figure source: *NVIDIA CUDA Programming Guide* ver. 3.0

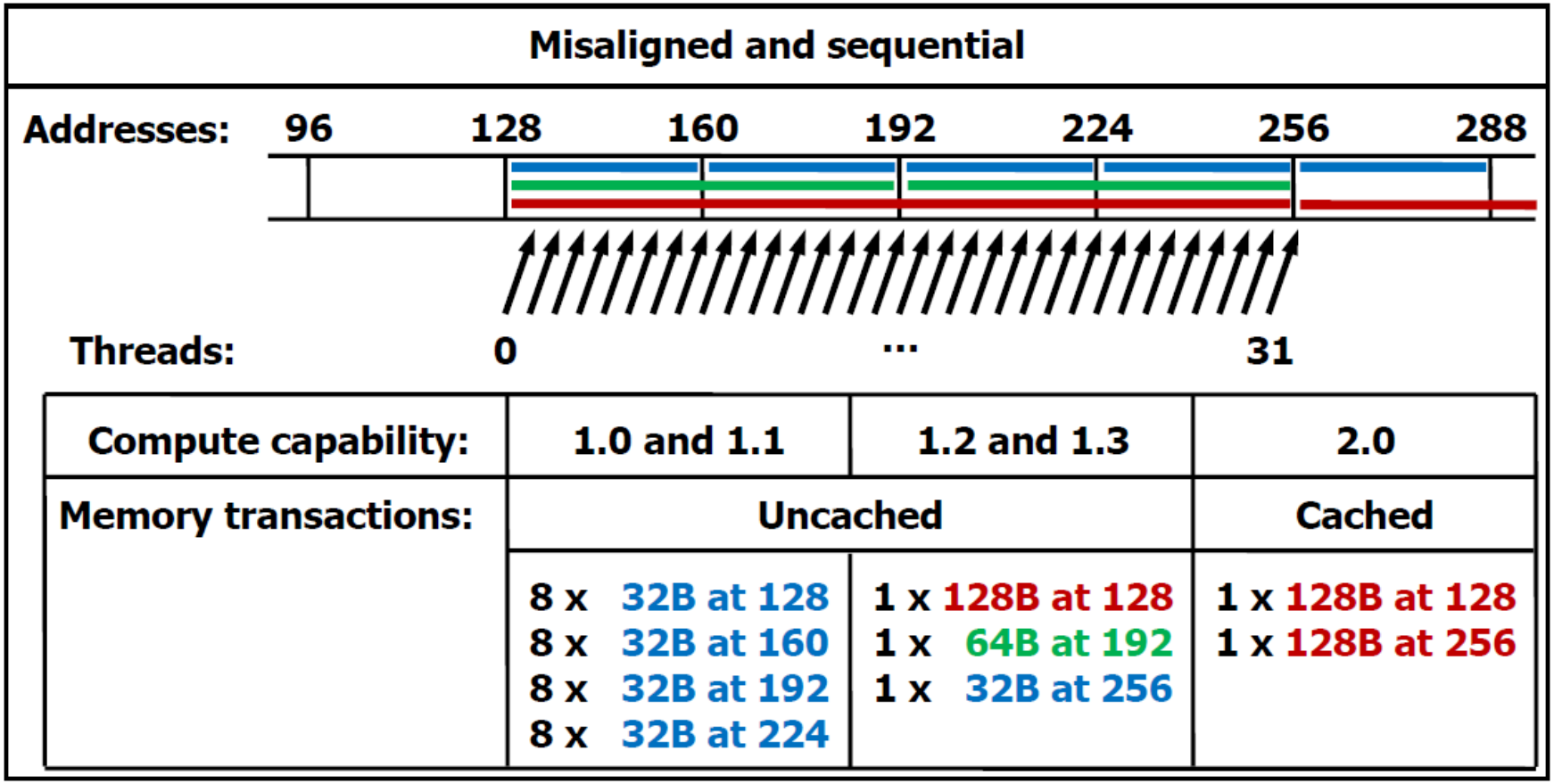
4-byte word per thread example



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Figure source: *NVIDIA CUDA Programming Guide* ver. 3.0

4-byte word per thread example



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

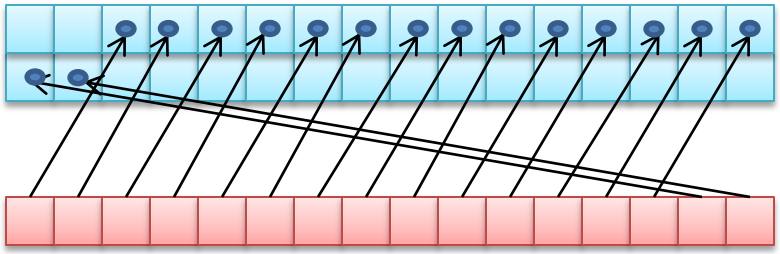
Figure source: *NVIDIA CUDA Programming Guide* ver. 3.0

Misaligned access

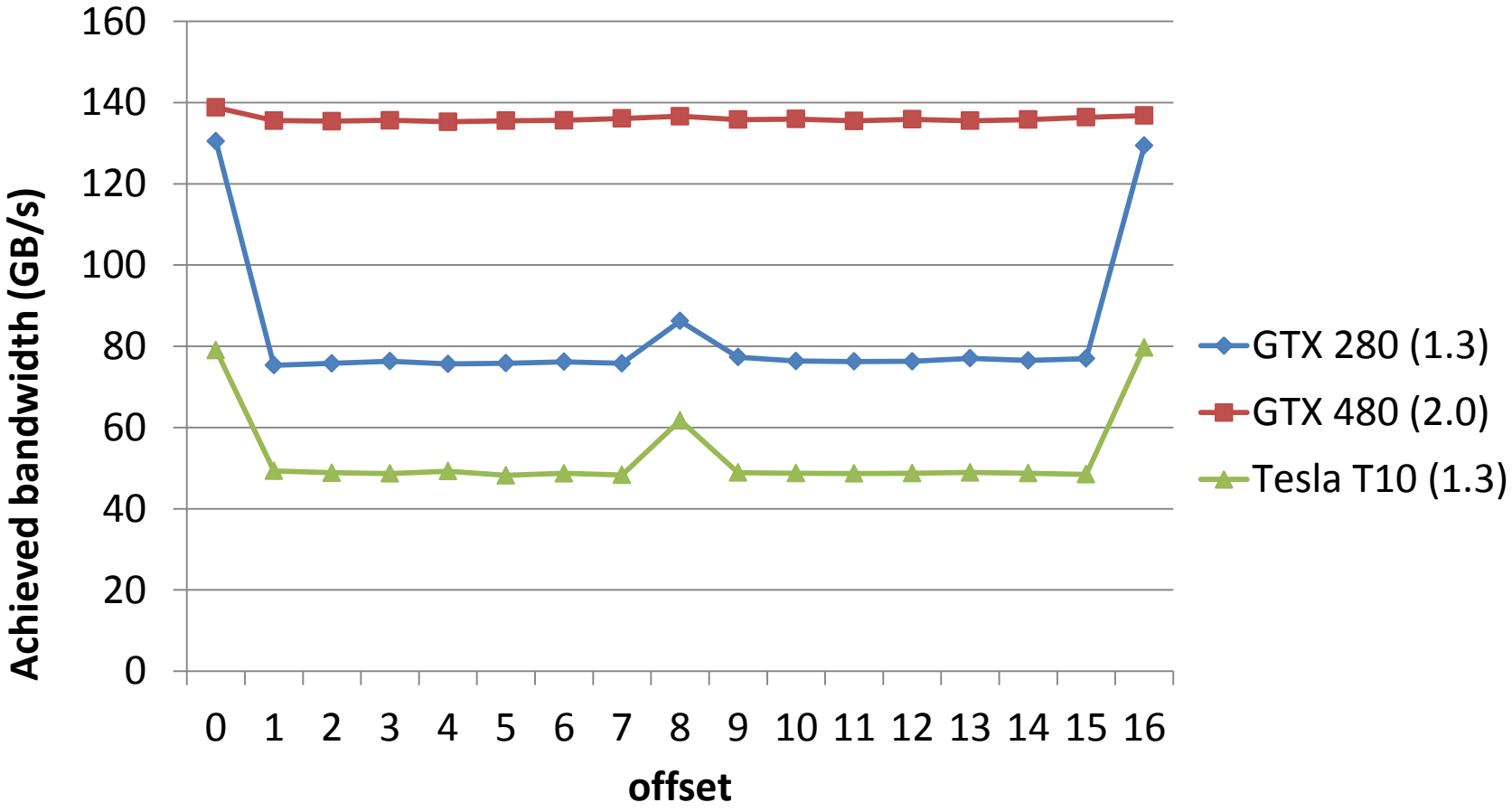
- Misalignment results in issuing multiple memory access instructions
- Example kernel

```
__global__  void offsetCopy(float* A, float* B, int offset)
{
   long int i = blockIdx.x * blockDim.x + threadIdx.x + offset;
   A[i] = B[i];
}
```

- 2 words offset example



Effects of misaligned access

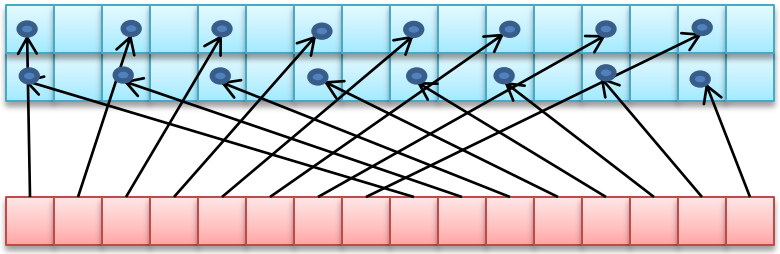


Strided access

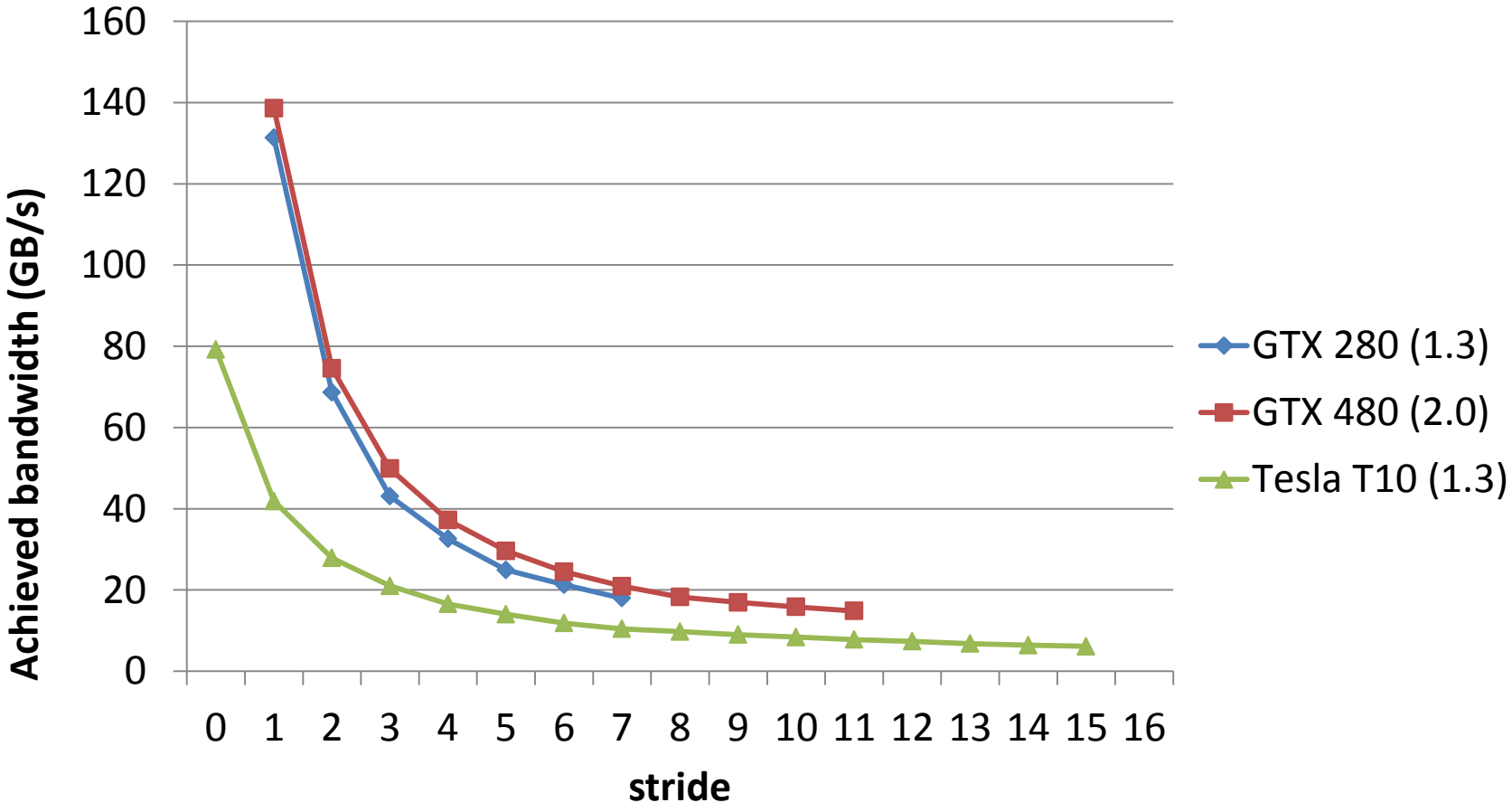
- Strided access results in issuing multiple memory access instructions
- Example kernel

```
__global__  void strideCopy(float* A, float* B, int stride)
{
   long int i = (blockIdx.x * blockDim.x + threadIdx.x) * stride;
   A[i] = B[i];
}
```

- Stride 2 example



Effects of strided assess



Local memory

- Resides in the device memory
- Allocated for per-thread access
- Allocated by the compiler to hold automatic variables when there is an insufficient register space
- As slow as global memory

Constant memory

- Resides in the device memory
- Cached
- As long as all threads in a half-warp read the same value from constant cache, the read is as fast as from a register
- Access to different addresses from the same half-warp is serialized, thus cost scales linearly with the number of unique memory locations accessed

- Example to copy host memory to constant memory

```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
```

Texture memory

- Resides in the device memory
- Cached
- Optimized for 2D spatial locality
  - Threads of the same warp that read texture addresses that are close together in 2D will achieve best performance
  - Addressing calculations are performed outside of the kernel by dedicated units

Shared memory

- Resides on-chip, thus much faster (~100x) than any off-chip memory
  - 16 KB per SM on pre-Fermi architecture
  - 16 or 48 KB per SM on Fermi architecture
- Divided into equally-sized memory modules (banks) that can be accessed simultaneously
  - Any memory read/write request made to n addresses that fall into n separate banks will be served simultaneously
  - If any two addresses are from the same memory bank, there is a bank conflict and the accesses will be serialized = access penalty

Shared memory

- For compute capability 1.x
  - 16 banks per SM
  - 32-bit wide banks
  - A shared memory request for a warp of threads is split into two accesses, each for a half-warp
    - Need to avoid bank conflicts in each half-warp
- For compute capability 2.0
  - 32 banks per SM on Fermi architecture
  - 32-bit wide banks
  - A shared memory request for a warp of threads is not split
    - Need to avoid bank conflicts in each warp

Register file

- 16KB per SM on compute capability 1.x
- 32 KB per SM on compute capability 2.0
- Registers are partitioned among concurrent threads scheduled on a given SM
  - Compiler and hardware scheduler are in charge of scheduling the use of registers to avoid bank conflicts
  - When not enough space in the register file, space will be allocated in the local memory for spill-over registers = expensive access

Dealing with register dependencies

- Register dependency arises when an instruction uses a result stored in a register written by an instruction before it
  - Latency is ~24 cycles
- To hide this latency, SM should be running a sufficiently large number of threads in other warps
  - At least 192 threads for compute capability 1.x
  - As many as 384 threads for compute capability 2.0
    - Registers are dual-issue on compute capability 2.0

Threads

- 32 Threads = 1 Warp
  - A warp (of threads) executes one common instruction at a time
- A “thread block” is a collection of warps that run on the same core and share a partition of local store
  - The number of warps in the thread block is configurable
  - Threads in a thread block start at the same instruction address and execute in parallel
- 32 max warps can be active per Warp Scheduler
  - 1024 threads active at once per Scheduler
  - Actual number of threads managed depends on amount of memory used per thread

Occupancy

- Ratio of the number of active warps per multiprocessor to the maximum number of possible active warps
  - Low occupancy results in inability to hide device memory access latency
- Occupancy is influenced by the number of thread blocks, number of threads per block, and by the register use

Occupancy calculator

CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	1.3	(Help)

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	8	
Shared Memory Per Block (bytes)	2048	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	**1024**	(Help)
Active Warps per Multiprocessor	**32**	
Active Thread Blocks per Multiprocessor	**4**	
Occupancy of each Multiprocessor	**100%**	

Physical Limits for GPU:	1.3
Threads / Warp	32
Warps / Multiprocessor	32
Threads / Multiprocessor	1024
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	16384
Register allocation unit size	512
Shared Memory / Multiprocessor (bytes)	16384
Warp allocation granularity (for register allocation)	2

Allocation Per Thread Block	
Warps	8
Registers	2048
Shared Memory	2048

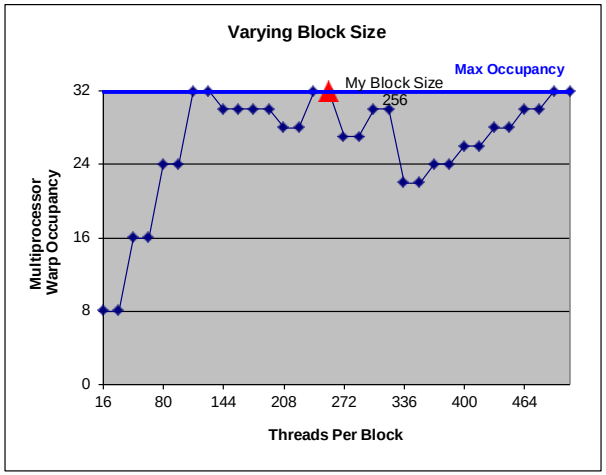
These data are used in computing the occupancy data in blue

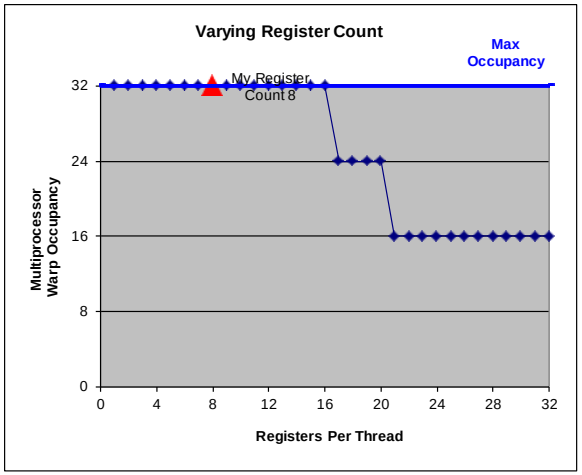
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	4
Limited by Registers / Multiprocessor	8
Limited by Shared Memory / Multiprocessor	8

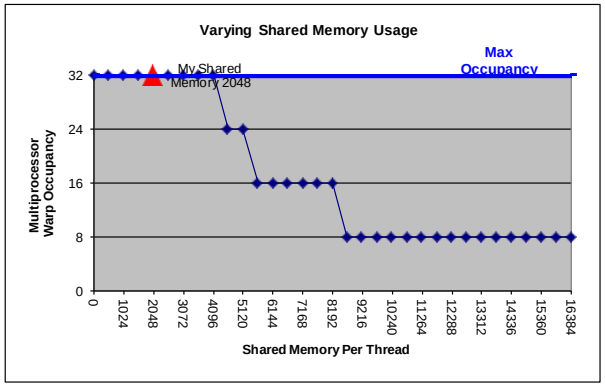
Thread Block Limit Per Multiprocessor highlighted RED

CUDA Occupancy Calculator	
Version:	1.5
Copyright and License	

Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.







Threads and blocks

- The number of blocks in a grid should be larger than the number of multiprocessors
  - Each multiprocessor should have at least one block to execute
  - Desirable to have multiple active blocks per multiprocessor to avoid entire multiprocessor waiting on __syncthreads()
  - Summary: thousands of grid blocks should be launched
- The number of threads per block should be selected to maximize the occupancy
  - 512 maximum threads per thread block
  - Occupancy also depends on the register usage as well
  - Threads per block should be a multiple of warp size
  - A minimum of 64 threads per block is desirable, but only if there are multiple concurrent blocks per SM
- Typically some experimentation is needed to find out best configuration

Threads Synchronization

- __syncthreads() synchronizes all threads in a thread block
  - Once all threads have reached this point, execution resumes normally
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- Should be used in conditional code only if the conditional is uniform across the entire thread block

Arithmetic instructions

- The hardware is designed for single precision floating point arithmetic
- Integer division and modulo operations are particularly costly -> use shift when possible
- Reciprocal square root: use rsqrtf() instead of 1.0f/sqrtf()
- Avoid automatic conversion between double an float
- Use the fast math libraries when possible, they start with prepended underscores __

Control flow

- Avoid different execution path within the same warp
  - Different execution paths in a single warp will be serialized
- Help compiler to do branch prediction
  - E.g., unroll loops with #pragma unroll

Final recommendations

- Select parallel algorithm instead of a sequential one
- Use the effective bandwidth as a measure of the optimization benefits
- Minimize data transfer between the host and device memory
- Ensure coalesced device memory access
- Minimize use of global memory
- Avoid execution path divergence

Final recommendations

- Avoid bank conflicts when accessing shared memory
- Use shared memory to avoid redundant data access to global memory
- Maintain at least 25% occupancy
- Have at least 32 threads per block
- Use fast math when possible

Porting matrix multiplier to CUDA

- cd ../tutorial/src3

- Compile & run CPU version

  icc -O3 mmult.c -o mmult

  ./mmult

1024.00 1024.00 1024.00 1024.00 1024.00 ...
1024.00 1024.00 1024.00 1024.00 1024.00 ...
1024.00 1024.00 1024.00 1024.00 1024.00 ...
1024.00 1024.00 1024.00 1024.00 1024.00 ...
1024.00 1024.00 1024.00 1024.00 1024.00 ...
...
msec = 2215478 GFLOPS = 0.969

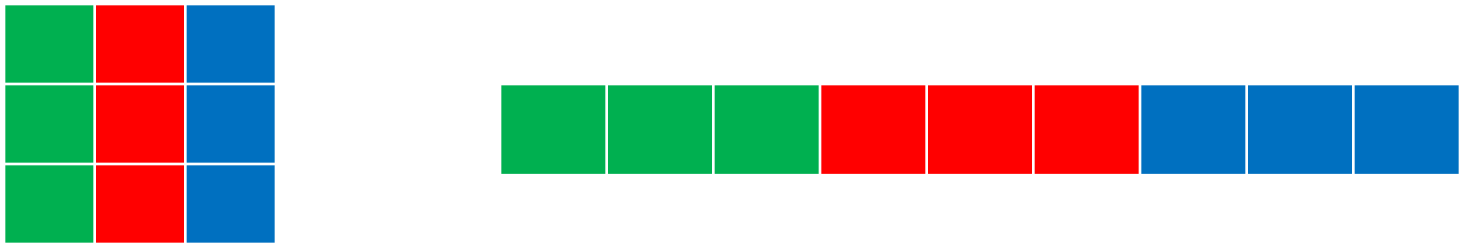
```
int main(int argc, char* argv[])
{
   int N = 1024;

   struct timeval t1, t2, ta, tb;
   long msec1, msec2;
   float flop, mflop, gflop;

   float *a = (float *)malloc(N*N*sizeof(float));
   float *b = (float *)malloc(N*N*sizeof(float));
   float *c = (float *)malloc(N*N*sizeof(float));

   minit(a, b, c, N);

   gettimeofday(&t1, NULL);
   mmult(a, b, c, N);  // a = b * c
   gettimeofday(&t2, NULL);

   mprint(a, N, 5);

   free(a);
   free(b);
   free(c);

   msec1 = t1.tv_sec * 1000000 + t1.tv_usec;
   msec2 = t2.tv_sec * 1000000 + t2.tv_usec;
   msec2 -= msec1;
   flop = N*N*N*2.0f;
   mflop = flop / msec2;
   gflop = mflop / 1000.0f;
   printf("msec = %10ld   GFLOPS = %.3f\n", msec2, gflop);
}
```

```
// a = b * c
void mmult(float *a, float *b, float *c, int N)
{
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         for (int i = 0; i < N; i++)
            a[i+j*N] += b[i+k*N]*c[k+j*N];
}


void minit(float *a, float *b, float *c, int N)
{
   for (int j = 0; j < N; j++)
      for (int i = 0; i < N; i++) {
         a[i+N*j] = 0.0f;
         b[i+N*j] = 1.0f;
         c[i+N*j] = 1.0f;
      }
}

void mprint(float *a, int N, int M)
{
   int i, j;

   for (int j = 0; j < M; j++)
   {
      for (int i = 0; i < M; i++)
         printf("%.2f ", a[i+N*j]);
      printf("...\n");
   }
   printf("...\n");
}
```

Matrix-matrix multiplication example (BLAS SGEMM)

```
for (i = 0; i < n; ++i)
    for (j = 0; j < m; ++j)
        for (k = 0; k < p; ++k)
            a[i+n*j] += b[i+n*k] * c[k+p*j];
```

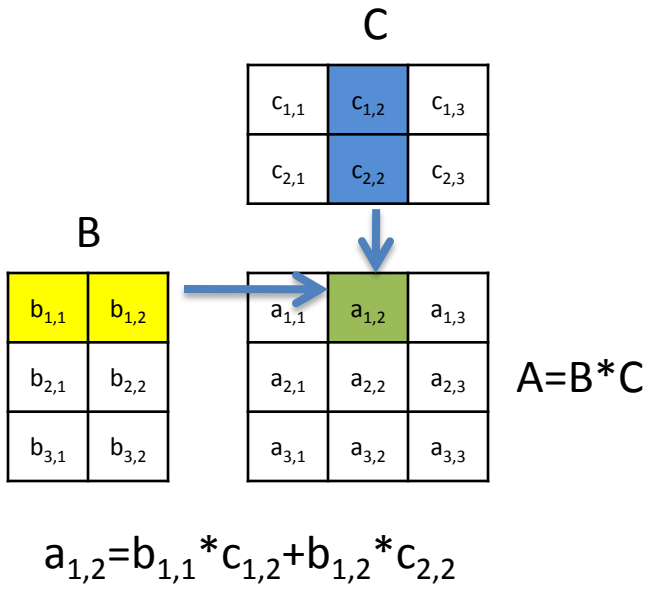
- Matrices are stored in column-major order



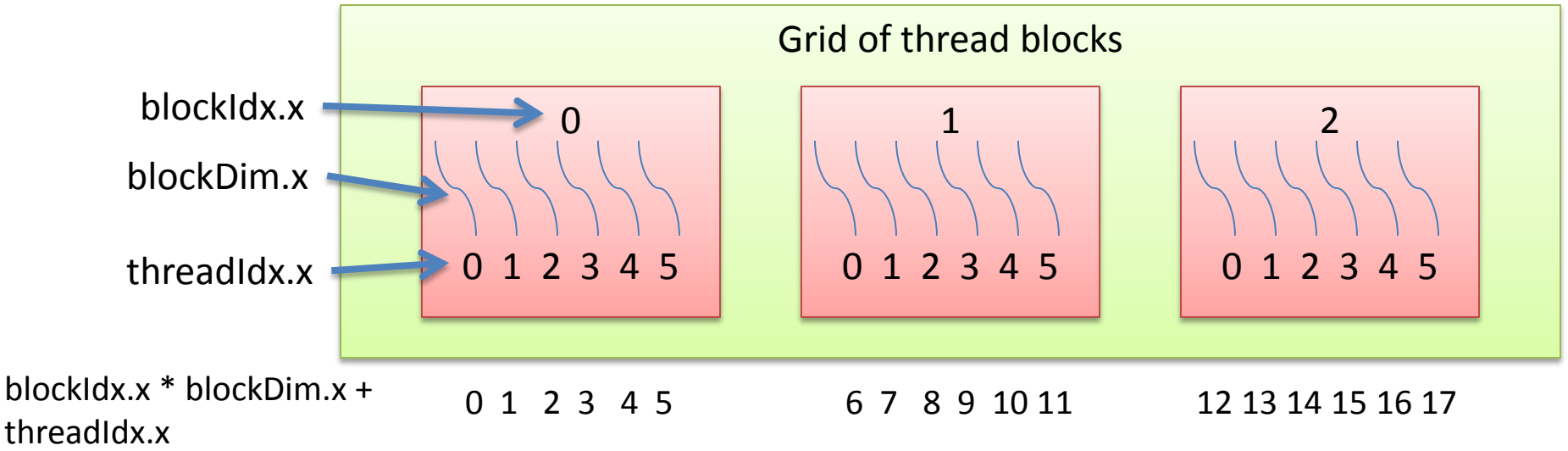
- For reference, jki-ordered version runs at 1.7 GFLOPS on 3 GHz Intel Xeon (single core)

Map this code:

```
for (i = 0; i < n; ++i)
    for (j = 0; j < m; ++j)
        for (k = 0; k < p; ++k)
            a[i+n*j] += b[i+n*k] * c[k+p*j];
```



into this (logical) architecture:



Strip-mined the stride-1 *i* loop to SIMD width of 32

```
for (i = 0; i < n; ++i)
   for (j = 0; j < m; ++j)
      for (k = 0; k < p; ++k)
         a[i+n*j] += b[i+n*k] * c[k+p*j];
```

```
for (is = 0; i < n; is+=32)
  for (i = is; i < is+32; ++i)
    for (j = 0; j < m; ++j)
      for (k = 0; k < p; ++k)
        a[i+n*j] += b[i+n*k] * c[k+n*j];
```

Run the *i* element as a thread block and the *is* strip loop and *j* loop in parallel

```
for (is = 0; i < n; is+=32)
  for (i = is; i < is+32; ++i)
    for (j = 0; j < m; ++j)
      for (k = 0; k < p; ++k)
        a[i+n*j] += b[i+n*k] * c[k+p*j];
```

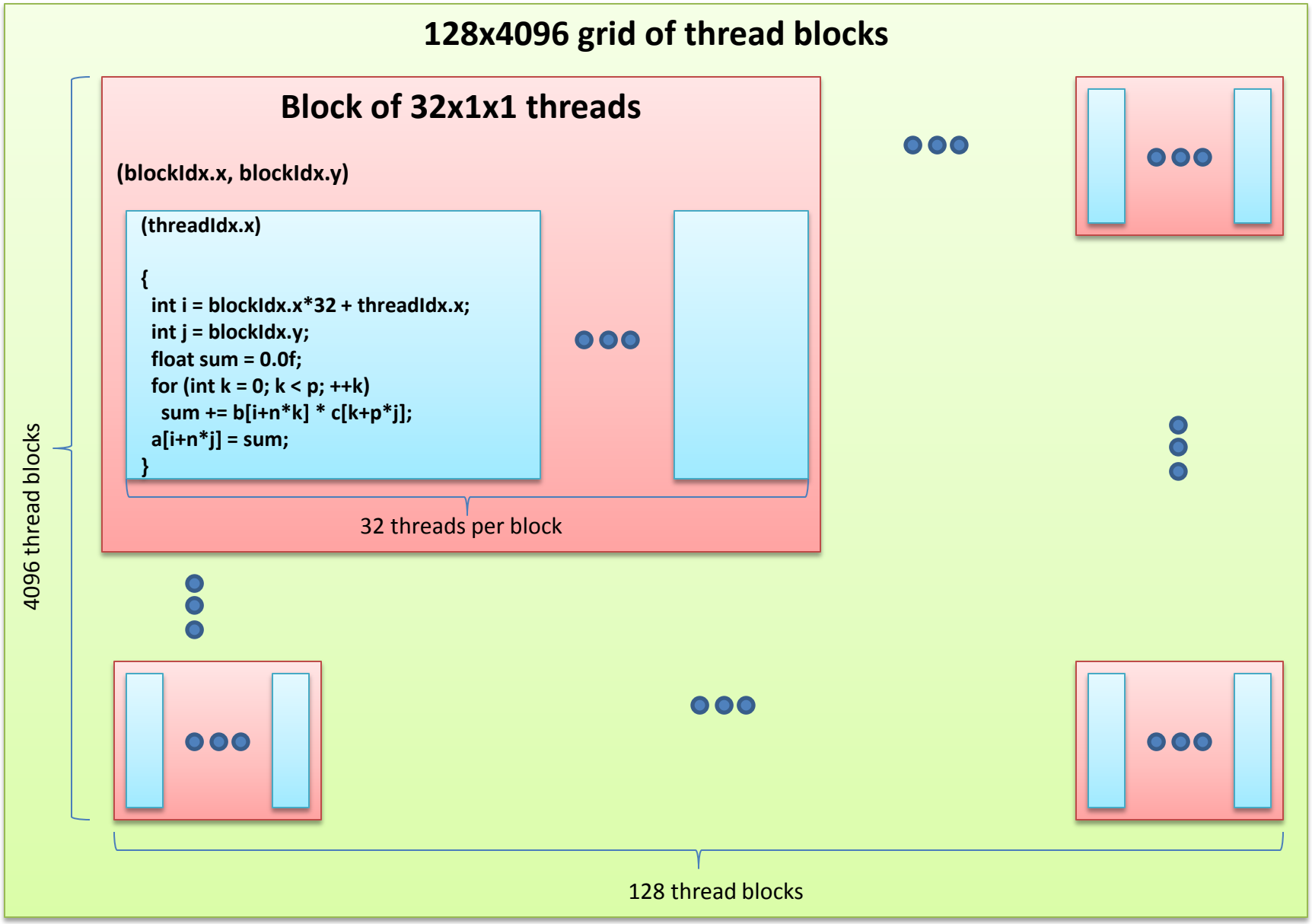
```
parfor (is = 0; i < n; is+=32)
  parfor (j = 0; j < m; ++j)
    SIMDfor (i = is; i < is+32; ++i)
      for (k = 0; k < p; ++k)
        a[i+n*j] += b[i+n*k] * c[k+p*j];
```

Parallel (grid) loops and SIMD (thread block) loop are handled implicitly by the GPU hardware

```
parfor (is = 0; i < n; is+=32)
  parfor (j = 0; j < m; ++j)
    SIMDfor (i = is; i < is+32; ++i)
      for (k = 0; k < p; ++k)
        a[i+n*j] += b[i+n*k] * c[k+p*j];
```

```
extern "C" __global__ void mmkernel (float* a, float* b, float* c, int n, int m,
int p)
{
   int i = blockIdx.x*32 + threadIdx.x;
   int j = blockIdx.y;
   float sum = 0.0f;
   for (int k = 0; k < p; ++k)  sum += b[i+n*k] * c[k+p*j];
   a[i+n*j] = sum;
}
```

dim3 threads (32);
dim3 grid(4096/32, 4096);



Version 1

```
int i = blockIdx.x*32 + threadIdx.x;
int j = blockIdx.y;
float sum = 0.0f;
for (int k = 0; k < p; ++k)
        sum += b[i+n*k] * c[k+p*j];
a[i+n*j] = sum;
```

[kindr@ac31 src5]$ ./mmult_gpu

matrix 4096x4096
grid 128x4096
block 32x1x1

msec = 5779620 GFLOPS = 23.780

Version 1

Threads per block (SIMD width)	Grid size	performance (GFLOPS)		
		Overall	kernel	
32	128x4096	23.3	24.2	One warp per thread block. Thus, at most only 8 thread blocks are active on each multiprocessor, out of 32 max.
64	64x4096	26.0	27.0	If 8 thread blocks are scheduled on each multiprocessor, we get up to 16 warps, so the multithreading is more efficient.
128	32x4096	24.5	25.3	
256	16x4096	24.9	25.9	

Strip-mine *k* loop and load a strip of *c* into the multiprocessor local memory

```
parfor (is = 0; i < n; is+=32)
  parfor (j = 0; j < m; ++j)
    SIMDfor (i = is; i < is+32; ++i)
      for (k = 0; k < p; ++k)
        a[i+n*j] += b[i+n*k] * c[k+p*j];
```

```
parfor (is = 0; i < n; is+=32)
  parfor (j = 0; j < m; ++j)
    SIMDfor (i = is; i < is+32; ++i)
      for (ks=0; ks<p; ks+=32)
        cb[ks:ks+31]=c[ks+p*j:ks+31+p*j];
        for (k = ks; k < ks+32; ++k)
          a[i+n*j] += b[i+n*k] * cb[k-ks];
```

Version 2

```
int tx = threadIdx.x;    int i = blockIdx.x*32 + tx;   int j = blockIdx.y;
__shared__ float cb[32];
float sum = 0.0f;
for (int ks = 0; ks < p; ks += 32)  {
  cb[tx] = c[ks+tx+p*j];
  for (int k = ks; k < ks+32; ++k) sum += b[i+n*k] * cb[k-ks];
}
a[i+n*j] = sum;
```

```
[kindr@ac31 src5]$ ./mmult_gpu

matrix 4096x4096
grid 128x4096
block 32x1x1

msec =    4538683   GFLOPS = 30.282
```

Version 2

Threads per block (SIMD width)	Grid size	performance (GFLOPS)		
		Overall	kernel	
32	128x4096	28.5	29.8	
64	64x4096	40.4	43.1	
128	32x4096	40.5	43.2	
256	16x4096	41.2	44.0	

Each kernel instance computes 2 values of the *i* loop

```
parfor (is = 0; i < n; is+=32)
  parfor (j = 0; j < m; ++j)
    SIMDfor (i = is; i < is+32; ++i)
      for (ks=0; ks<p; ks+=32)
        cb[ks:ks+31]=c[ks+p*j:ks+31+p*j];
        for (k = ks; k < ks+32; ++k)
          a[i+n*j] += b[i+n*k] * cb[k-ks];
```

```
parfor (is = 0; i < n; is+=64)
  parfor (j = 0; j < m; ++j)
    SIMDfor (i = is; i < is+32; ++i)
      for (ks=0; ks<p; ks+=32)
        cb[ks:ks+31]=c[ks+p*j:ks+31+p*j];
        for (k = ks; k < ks+32; ++k)
          a[i+n*j] += b[i+n*k] * cb[k-ks];
          a[i+32+n*j] += b[i+32+n*k] * cb[k-ks];
```

Version 3

```
int tx = threadIdx.x;   int i = blockIdx.x*64 + tx;   int j = blockIdx.y;
__shared__ float cb[32];
float sum0 = 0.0f, sum1 = 0.0f;
for (int ks = 0; ks < p; ks += 32) {
  cb[tx] = c[ks+tx+p*j];
  __syncthreads();
  for (int k = ks; k < ks+32; ++k) {  sum0 += b[i+n*k] * cb[k-ks];  sum1 += b[i+32+n*k] * cb[k-ks];  }
  __syncthreads();
}
a[i+n*j] = sum0;
a[i+32+n*j] = sum1;
```

[kindr@ac31 src5]$ ./mmult_gpu

matrix 4096x4096
grid 64x4096
block 32x1x1

msec = 3182941 GFLOPS = 43.180

Version 3

x2

Threads per block (SIMD width)	Grid size	performance (GFLOPS)		
		overall	kernel	
32	64x4096	40.4	43.1	
64	32x4096	40.7	43.4	
128	16x4096	40.8	43.6	

x4

Threads per block (SIMD width)	Grid size	performance (GFLOPS)		
		overall	kernel	
32	32x4096	40.8	43.4	
64	16x4096	40.9	43.6	
128	8x4096	39.6	42.1	

Each kernel instance computes 2 values of the *j* loop

```
parfor (is = 0; i < n; is+=32)
  parfor (j = 0; j < m; ++j)
    SIMDfor (i = is; i < is+32; ++i)
      for (ks=0; ks<p; ks+=32)
        cb[ks:ks+31]=c[ks+p*j:ks+31+p*j];
        for (k = ks; k < ks+32; ++k)
          a[i+n*j] += b[i+n*k] * cb[k-ks];
```

```
parfor (is = 0; i < n; is+=32)
  parfor (j = 0; j < m; j+=2)
    SIMDfor (i = is; i < is+32; ++i)
      for (ks=0; ks<p; ks+=32)
        cb0[ks:ks+31]=c[ks+p*j:ks+31+p*j];
        cb1[ks:ks+31]=c[ks+p*(j+1):ks+31+p*(j+1)];
        for (k = ks; k < ks+32; ++k)
          a[i+n*j] += b[i+n*k] * cb0[k-ks];
          a[i+32+n*(j+1)] += b[i+32+n*k] * cb1[k-ks];
```

Version 4

```
int tx = threadIdx.x;    int i = blockIdx.x*32 + tx;    int j = blockIdx.y*2;
__shared__ float cb0[32], cb1[32];
float sum0 = 0.0f, sum1 = 0.0f;
for (int ks = 0; ks < p; ks += 32) {
  cb0[tx] = c[ks+tx+p*j];
  cb1[tx] = c[ks+tx+p*(j+1)];
  __syncthreads();
  for (int k = ks; k < ks+32; ++k) { float rb = b[i+n*k];  sum0 += rb * cb0[k-ks];  sum1 += rb * cb1[k-ks]; }
  __syncthreads();
}
a[i+n*j] = sum0;
a[i+n*(j+1)] = sum1;
```

[kindr@ac31 src5]$ ./mmult_gpu

matrix 4096x4096
grid 128x2048
block 32x1x1

msec =   2335358   GFLOPS = 58.851

Version 4

x2

Threads per block (SIMD width)	Grid size	performance (GFLOPS)		
		overall	kernel	
32	128x2048	52.7	57.3	
64	64x2048	75.2	84.8	
128	32x2048	76.2	86.3	

x4

Threads per block (SIMD width)	Grid size	performance (GFLOPS)		
		overall	kernel	
32	128x1024	92.3	107.4	
64	64x1024	131.3	163.8	
128	32x1024	134.6	169.1	

Bottom line

- It is easy enough to get something to run on a GPU
- But it is difficult to get it to run fast
  - Things to consider
    - which algorithm to use; some algorithms are better suited for GPUs than others
    - understand if the kernel is compute-bound or memory bandwidth bound and optimize it accordingly