Introduction to CUDA Programming

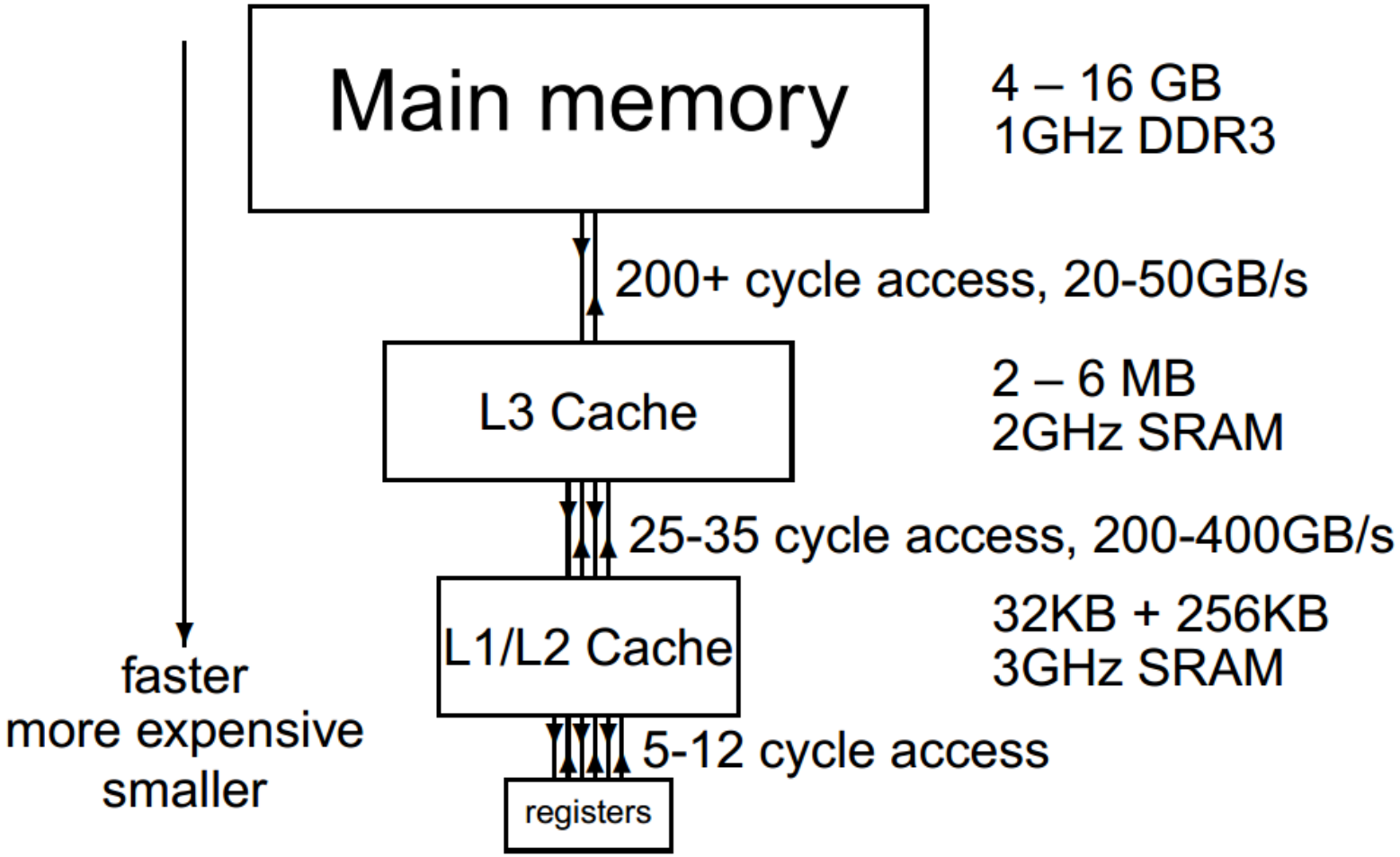
Lecture 2: different memory and variable types

高性能计算机研究中心

Memory

- Key challenge in modern computer architecture
  - no point in blindingly fast computation if data can't be moved in and out fast enough
  - need lots of memory for big applications
  - very fast memory is also very expensive
  - end up being pushed towards a hierarchical design

CPU Memory Hierarchy



Memory Hierarchy

- Execution speed relies on exploiting data *locality*
  - temporal locality: a data item just accessed is likely to be used again in the near future, so keep it in the cache
  - spatial locality: neighbouring data is also likely to be used soon, so load them into the cache at the same time using a 'wide' bus (like a multi-lane motorway)
- This wide bus is only way to get high bandwidth to slow main memory

Caches

- The cache line is the basic unit of data transfer; typical size is 64 bytes = 8×8-byte items.

- With a single cache, when the CPU loads data into a register:
  - it looks for line in cache
  - if there (hit), it gets data
  - if not (miss), it gets entire line from main memory, displacing an existing line in cache (usually least recently used)
- When the CPU stores data from a register:
  - same procedure

Importance of Locality

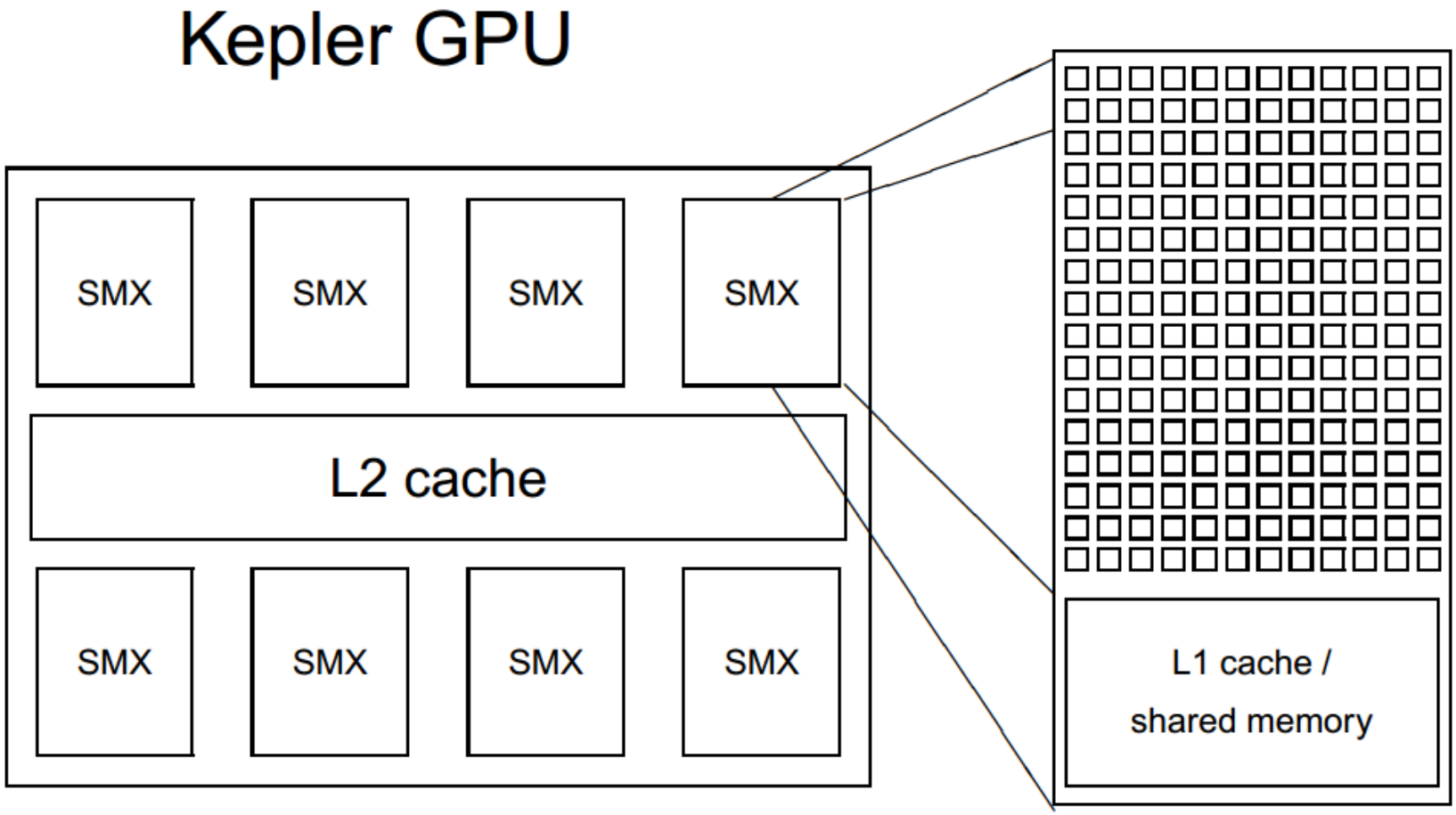
- Typical workstation:
  10 Gflops CPU
  20 GB/s memory ←→ L2 cache bandwidth
  64 bytes/line
  20GB/s = 300M line/s = 2.4G double/s

- At worst, each flop requires 2 inputs and has 1 output, forcing loading of 3 lines ⇒ 100 Mflops

- If all 8 variables/line are used, then this increases to 800 Mflops.

- To get up to 10Gflops needs temporal locality, re-using data already in the cache.

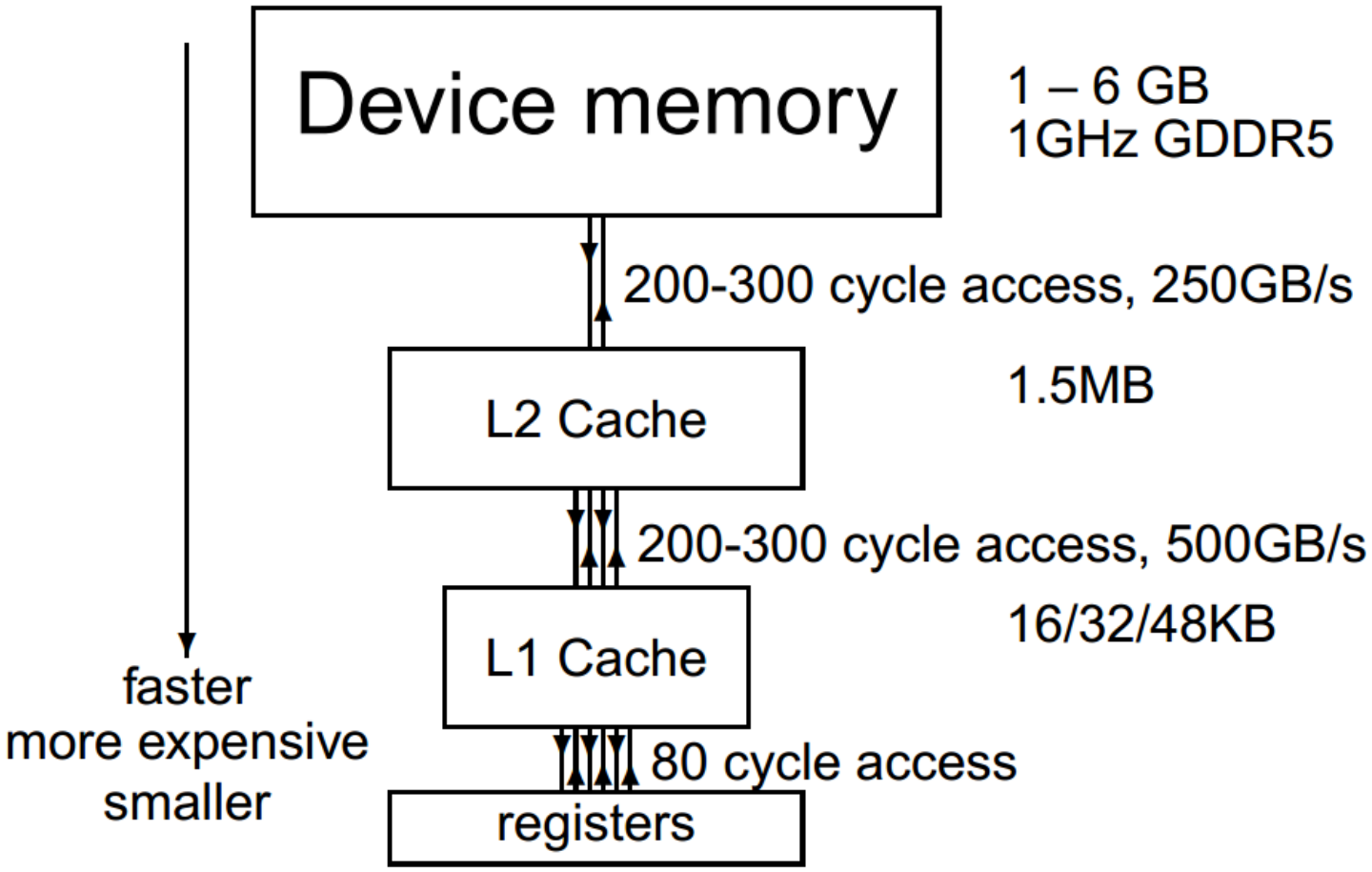
Kepler


Kepler GPU
SMX
SMX
SMX
SMX
L2 cache
SMX
SMX
SMX
SMX
L1 cache /
shared memory

Kepler

- usually 128 bytes cache line (32 floats or 16 doubles) (32 bytes under certain circumstances)
- 384-bit memory bus from device memory to L2 cache
- up to 250 GB/s bandwidth
- unified 1.5MB L2 cache for all SMX's
- each SMX has 64kB of shared memory / L1 cache (split 16/48, 32/32 or 48/16)
- no global cache coherency as in CPUs, so should (almost) never have different blocks updating the same global array elements

GPU Memory Hierarchy



Importance of Locality

- 1Tflops GPU
  250 GB/s memory ←→ L2 cache bandwidth
  128 bytes/line
  250GB/s = 2G line/s = 32G double/s

- At worst, each flop requires 2 inputs and has 1 output, forcing loading of 3 lines ⇒ 670 Mflops
- If all 16 doubles/line are used, increases to 11 Gflops
- To get up to 500Gflops needs about 15 flops per double transferred to/from device memory

- Even with careful implementation, many algorithms are bandwidth-limited not compute-bound

Practical 1 kernel

```
__global__ void my_first_kernel(float *x)
{
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  x[tid] = threadIdx.x;
}
```

- **32 threads in a warp will address neighbouring elements of array** x
- **if the data is correctly "aligned" so that** x[0] **is at the beginning of a cache line, then** x[0]– x[31] **will be in same cache line – a "coalesced" transfer**
- **hence we get perfect spatial locality**

A bad kernel

```
__global__ void bad_kernel(float *x)
{
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  x[1000 * tid] = threadIdx.x;
}
```

- in this case, different threads within a warp access widely spaced elements of array x – a “strided” array access
- each access involves a different cache line, so performance will be awful

Global arrays

- So far, concentrated on global / device arrays:
  - held in the large device memory
  - allocated by host code
  - pointers held by host code and passed into kernels
  - continue to exist until freed by host code
  - since blocks execute in an arbitrary order, if one block modifies an array element, no other block should read or write that same element

Global arrays

- Global variables can also be created by declarations with global scope within kernel code file

```
__device__ int reduction_lock=0;

__global__ void kernel_1(...) {
...
}

__global__ void kernel_2(...) {
...
}
```

Global arrays

- **the** __device__ **prefix tells** nvcc **this is a global variable in the GPU, not the CPU.**
- **the variable can be read and modified by any kernel**
- **its lifetime is the lifetime of the whole application**
- **can also declare arrays of fixed size**
- **can read/write by host code using special routines** cudaMemcpyToSymbol, cudaMemcpyFromSymbol **or with standard** cudaMemcpy **in combination with** cudaGetSymbolAddress
- **in my own CUDA programming, I rarely use this capability but it is occasionally very useful**

Constant variables

- **Very similar to global variables, except that they can't be modified by kernels:**
  - **defined with global scope within the kernel file using the prefix** __constant__
  - **initialised by the host code using** cudaMemcpyToSymbol**,** cudaMemcpyFromSymbol **or** cudaMemcpy **in combination with** cudaGetSymbolAddress
  - **I use it all the time in my applications; practical 2 has an example**

Constant variables

- Only 64KB of constant memory, but big benefit is that each SMX has a 8KB cache
  - when all threads read the same constant, almost as fast as a register
  - doesn't tie up a register, so very helpful in minimising the total number of registers required

Constants

- A constant variable has its value set at run-time
- But code also often has plain constants whose value is known at compile-time:

```
#define PI 3.1415926f
a = b / (2.0f * PI);
```

- Leave these as they are – they are embedded into the executable code so they don't use up any registers
- Don't forget the f at the end if you want single precision; in C/C++

single × double= double

Registers

- Within each kernel, by default, individual variables are assigned to registers:

```
__global__ void lap(int I, int J, float *u1, float *u2) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j = threadIdx.y + blockIdx.y * blockDim.y;
  int id = i + j * I;

  if (i == 0 || i == I-1 || j == 0 || j == J-1) {
    u2[id] = u1[id]; // Dirichlet b.c.’s
  }
  else {
    u2[id] = 0.25f * (u1[id - 1] + u1[id + 1]
                      + u1[id - I] + u1[id + I]);
  }
}
```

Registers

- 64K 32-bit registers per SMX
- up to 63 registers per thread (up to 255 for K20 / K40)
- up to 2048 threads (at most 1024 per thread block)
- max registers per thread ⇒ 1024 threads (256 threads for K20 / K40)
- max threads ⇒ 32 registers per thread
- not much difference between “fat” and “thin” threads (except for K20 / K40)

Registers

- What happens if your application needs more registers?
- They “spill” over into L1 cache, and from there to device memory
- Precise mechanism unclear, probably the contents of some registers get “saved” to device memory so they can be used for other purposes, then the data gets “restored” later
- Anyway, the application suffers from the latency and bandwidth implications of using device memory

Registers

- Avoiding register spill is now one of my main concerns in big applications, but remember:
  - with 1024 threads, 400-600 cycle latency of device memory is usually OK because some warps can do useful work while others wait for data
  - provided there are 20 flops per variable read from (or written to) device memory, the bandwidth is not a limiting issue

Local arrays

- What happens if your application uses a little array?

```
__global__ void lap(float *u) {
  float ut[3];

  int tid = threadIdx.x + blockIdx.x * blockDim.x;

  for (int k = 0; k < 3; k++)
    ut[k] = u[tid + k * gridDim.x * blockDim.x];

  for (int k = 0; k < 3; k++)
    u[tid + k * gridDim.x * blockDim.x] =
      A[3 * k] * ut[0] + A[3 * k + 1] * ut[1] + A[3 * k + 2] * ut[2];
}
```

Local arrays

- In simple cases like this (quite common) compiler converts to scalar registers:

```
__global__ void lap(float *u) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  float ut0 = u[tid + 0 * gridDim.x * blockDim.x];
  float ut1 = u[tid + 1 * gridDim.x * blockDim.x];
  float ut2 = u[tid + 2 * gridDim.x * blockDim.x];

  u[tid + 0*gridDim.x*blockDim.x] = A[0]*ut0 + A[1]*ut1 + A[2]*ut2;
  u[tid + 1*gridDim.x*blockDim.x] = A[3]*ut0 + A[4]*ut1 + A[5]*ut2;
  u[tid + 2*gridDim.x*blockDim.x] = A[6]*ut0 + A[7]*ut1 + A[8]*ut2;
}
```

Local arrays

- In more complicated cases, it puts the array into device memory
  - still referred to in the documentation as a “local array” because each thread has its own private copy
  - held in L1 cache by default, may never be transferred to L2 cache or device memory
  - 16kB of L1 cache equates to 4096 32-bit variables, which is only 4 per thread when using 1024 threads
  - beyond this, it will have to spill to device memory

Shared memory

- **In a kernel, the prefix** __shared__ **as in**

```
__shared__ int x_dim;
__shared__ float x[128];
```

declares data to be shared between all of the threads in the thread block – any thread can set its value, or read it.

- **There can be several benefits:**
  - **essential for operations requiring communication between threads (e.g. summation in lecture 4)**
  - **useful for data re-use (I use it for unstructured grid applications)**
  - **alternative to local arrays in device memory**
  - **reduces use of registers when a variable has same value for all threads**

Shared memory

- **If a thread block has more than one warp, it's not pre-determined when each warp will execute its instructions – warp 1 could be many instructions ahead of warp 2, or well behind.**

- **Consequently, almost always need thread synchronisation to ensure correct use of shared memory.**

- **Instruction**

  __syncthreads();

  inserts a "barrier"; no thread/warp is allowed to proceed beyond this point until the rest have reached it (like a roll call on a school outing)

Shared memory

- So far, have discussed statically-allocated shared memory – the size is known at compile-time
- Can also create dynamic shared-memory arrays but this is more complex
- Total size is specified by an optional third argument when launching the kernel:

```
kernel<<<blocks, threads, shared_bytes>>>(...)
```

- Using this within the kernel function is complicated/tedious; see B.2.3 in Programming Guide

Shared memory

- **Kepler has 64KB which is split 16/48, 32/32 or 48/16 between L1 cache and shared memory:**
  - **this split can be set by the programmer using** cudaFuncSetCacheConfig **or** cudaDeviceSetCacheConfig
  - **default is 48KB of shared memory if not set by** cudaDeviceSetCacheConfig
  - **might be good to switch to 16KB of shared memory if the kernel doesn't need much shared memory**

Texture memory

- **Finally, we have texture memory:**
  - **originally, intended primarily for pure graphics applications**
  - **in Kepler K20/K40, the texture cache is 48kB and can be used as a cache for read-only global arrays which are accessed non-uniformly (i.e. different threads read different elements)**
  - **need to declare global array with**

    const __restrict__

    qualifiers so that the compiler knows that it is read-only

Non-blocking loads/stores

- What happens with the following code?

```
kernel<<<blocks, threads, shared_bytes>>>(...)
__kernel void lap(float *u1, float *u2) {
  float a;

  a = u1[threadIdx.x + blockIdx.x * blockDim.x]
  ...
  ...
  u2[threadIdx.x + blockIdx.x * blockDim.x] = a;
  ...
  ...
}
```

- Load doesn't block until needed; store doesn't block unless, or until, danger of modification

Active blocks per SMX

- Each block require certain resources:
  - threads
  - registers (registers per thread × number of threads)
  - shared memory (static + dynamic)
- Together these determine how many blocks can be run simultaneously on each SMX – up to a maximum of 16 blocks

Active blocks per SMX

- My general advice:
  - number of active threads depends on number of registers each needs
  - good to have at least 4 active blocks, each with at least 128 threads
  - smaller number of blocks when each needs lots of shared memory
  - larger number of blocks when they don't need shared memory

Active blocks per SMX

- On Kepler:
  - maybe 4 big blocks (256 threads) if each needs a lot of shared memory
  - maybe 8 small blocks (128 threads) if no shared memory needed
  - or 4 small blocks (128 threads) if each thread needs lots of registers

- Very important to experiment with different block sizes to find what gives the best performance.

Summary

- dynamic device arrays
- static device variables / arrays
- constant variables / arrays
- registers
- spilled registers
- local arrays
- shared variables / arrays
- textures

Key reading

- CUDA Programming Guide, version 5.5:
  - Appendix B.2, B.4 – essential
  - Chapter 3, sections 3.2.1-3.2.3

- Other reading:
  - Wikipedia article on caches:en.wikipedia.org/wiki/CPUcache
  - web article on caches: lwn.net/Articles/252125/
  - "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System": portal.acm.org/citation.cfm?id=1637764