Lecture 11

CUDA LIBRARIES

Linear algebra on the GPU

- Linear algebra on the CPU: BLAS, LAPACK
- GPU analogues: CUBLAS, MAGMA
- CUSPARSE library for sparse matrices
- Use of highly optimised libraries is usually better than writing your own code
- Writing efficient GPU code requires special care and understanding the peculiarities of underlying hardware

CUBLAS

- Implementation of BLAS (Basic Linear Algebra Subprograms) on top of CUDA
- Included with CUDA (hence free)
- Workflow:
  1. allocate vectors and matrices in GPU memory
  2. fill them with data
  3. call sequence of CUBLAS functions
  4. transfer results from GPU memory to host
- Helper functions to transfer data to/from GPU provided

Error checks

- In following example most error checks were removed for clarity
- Each CUBLAS function returns a status object containing information about possible errors
- It’s possible to catch errors via calls like this:

```
if (status != CUBLAS_STATUS_SUCCESS) {
print diagnostic information and exit}
```

Initialize program

```
#include <cuda.h> /* CUDA runtime API */
#include <cstdio>
#include <cublas_v2.h>
int main(int argc, char *argv[])
{
  float *x_host, *y_host; /* arrays for computation on host*/
  float *x_dev, *y_dev;   /* arrays for computation on device */
  int n = 32*1024*1024;
  float alpha = 0.5f;
  int nerror;
  size_t memsize;
  int i;
  /*  could add device detection here */
  memsize = n * sizeof(float);
```

Allocate memory on host and device

```
   /* allocate arrays on host */
  x_host = (float *)malloc(memsize);
  y_host = (float *)malloc(memsize);
/* allocate arrays on device */
  cudaMalloc((void **) &x_dev, memsize);
  cudaMalloc((void **) &y_dev, memsize);
  /* initialize arrays on host */
  for ( i = 0; i < n; i++)
  {
     x_host[i] = rand() / (float)RAND_MAX;
     y_host[i] = rand() / (float)RAND_MAX;
  }
  /* copy arrays to device memory (synchronous) */
  cudaMemcpy(x_dev, x_host, memsize, cudaMemcpyHostToDevice);
  cudaMemcpy(y_dev, y_host, memsize, cudaMemcpyHostToDevice);
```

Call CUBLAS function

```
cublasHandle_t handle;
cublasStatus_t status;
status  = cublasCreate(&handle);
int stride = 1;
status = cublasSaxpy(handle,n,&alpha,x_dev,stride,y_dev,stride);
/* check if cublasSaxpy launched succesfully */
if (status != CUBLAS_STATUS_SUCCESS)
 {
   printf ("Error in launching CUBLAS routine \n");
   exit (20);
 }
status = cublasDestroy(handle);
```

Retrieve computed data and finish

```
  /* retrieve results from device (synchronous) */
  cudaMemcpy(y_host, y_dev, memsize, cudaMemcpyDeviceToHost);
 /* ensure synchronization (cudaMemcpy is synchronous in most cases, but not all) */
  cudaDeviceSynchronize();
  /* use data in y_host*/
  /* free memory */
  cudaFree(x_dev);
  cudaFree(y_dev);
  free(x_host);
  free(y_host);
  return 0;
}
```

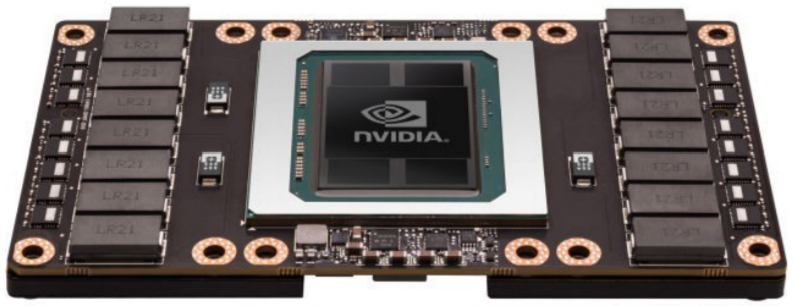
CUBLAS exercise

- Time the cublasSaxpy() operation in the CUBLAS version of the SAXPY code, ~syam/CSE746/saxpy/saxpy_cublas.cu (use GPU based timers, from saxpy_cuda_timed.cu).
- Compare that to the timing of the CUDA version, saxpy_cuda_timed.cu .

NEW CUDA FEATURES

From Fermi to Pascal

- Important HPC GPU generations (https://en.wikipedia.org/wiki/Nvidia_Tesla):
  - Fermi: 2010
  - Kepler: 2012
  - **Pascal**: 2016
  - **Volta**: 2017
  - **Turing**: 2018
  - **Ampere**: May 2020
  - Hopper: October 2022



Evolutionary changes

Specification	*Fermi*	*Pascal*
CUDA cores:	448	3584
SP flops:	1.03 TFlops	9.3 TFlops
Device memory:	5.2 GB	12 GB
Memory bandwidth:	148 GB/s	549 GB/s

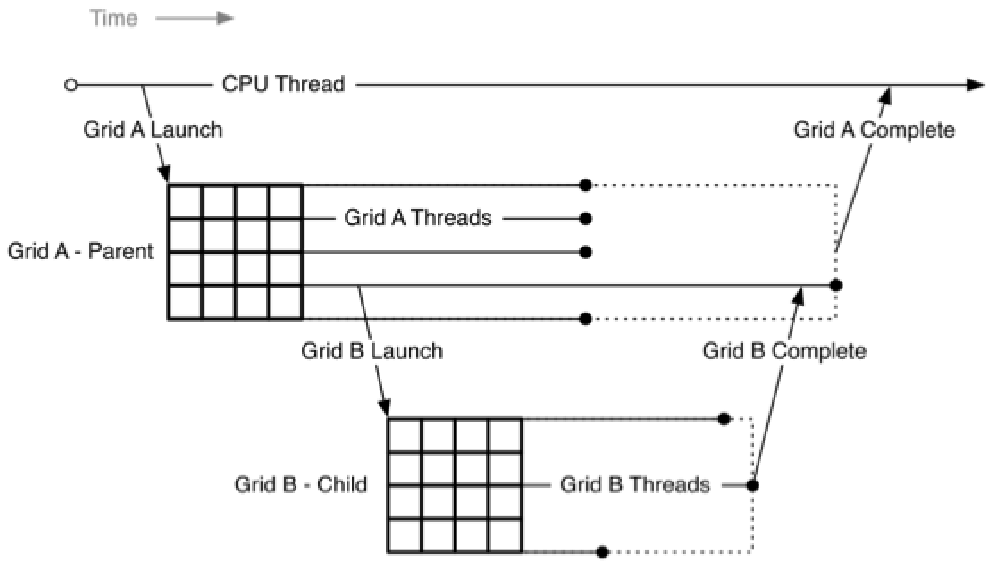
Revolutionary changes

- CUDA Dynamic Parallelism (CDP): new hard/software feature allowing for dynamic workload generation on GPU (kernels launched from kernels). Makes GPU much more general purpose computing device. First appeared in Kepler GPUs.
- Hyper-Q / MPS (multi-process service): in previous generations, multiple CPU threads could only access the GPU sequentially (one queue); Kepler and later expand that to 16+ parallel queues. This should significantly accelerate mixed MPI/CUDA and OpenMP/CUDA codes, without any code modifications. Also great for GPU farming.

Dynamic Parallelism

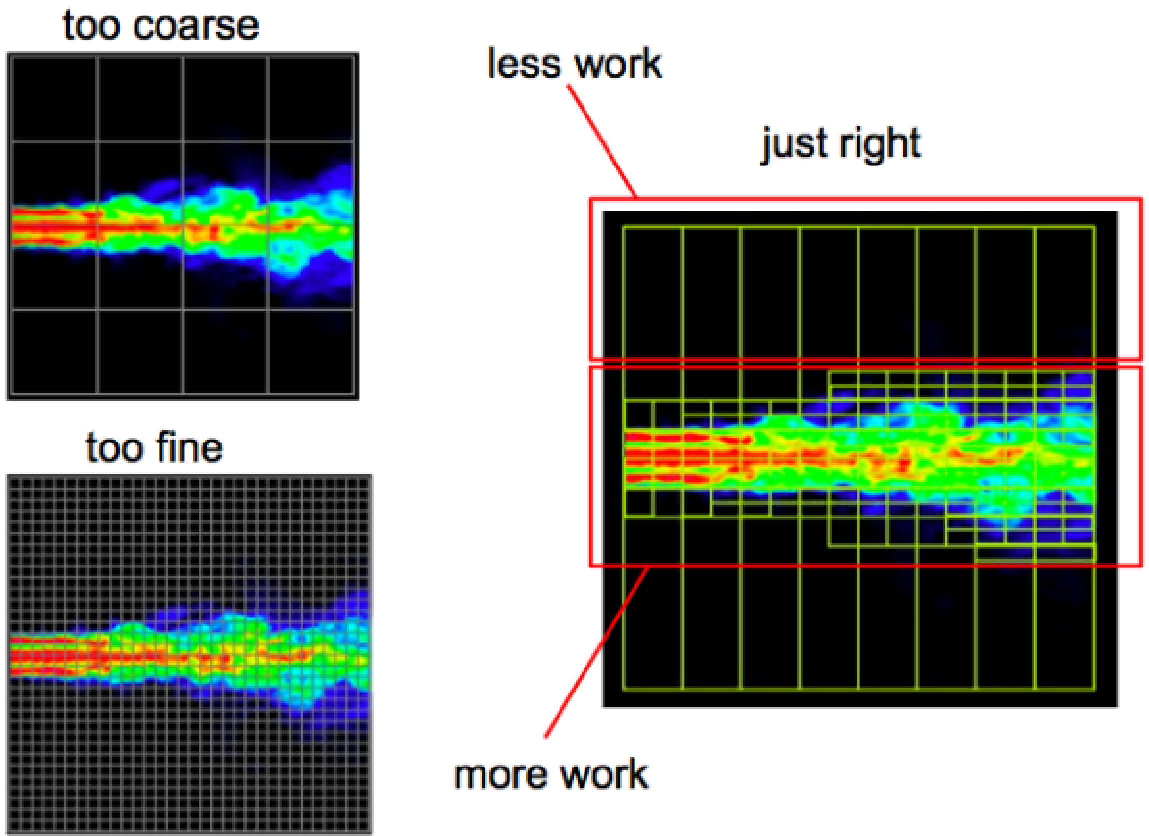
Dynamic Parallelism

- Dynamic parallelism (DP) is available in CUDA 5.0 and later on devices of Compute Capability 3.5 or higher (sm_35 for Kepler; sm_60 for Pascal).
- Under DP, an application can launch a coarse-grained kernel which in turn launches finer-grained kernels to do work where needed.



Dynamic Parallelism

- DP is perfect for adaptive grid codes and codes with recursion.



DP: simple example

- DP allows one to move almost everything to GPU.

```
// On device:
// Second level kernels (multi-threaded):
__global__  void  kernel1 (){}
__global__  void  kernel2 (){}

// Top level kernel (single-threaded):
__global__ void main_kernel (){
  if (threadIdx.x == 0) {
//  These second level kernels will run sequentially (would need streams for
concurrency)
     kernel1<<<Nblocks, Nthreads>>>();
     kernel2<<<Nblocks, Nthreads>>>();
     ...
     }}
// On host:
int main() {
   main_kernel<<<1,1>>>();}
```

Dynamic parallelism

- Permits kernel launches from inside kernels
- Kernels no longer have to be launched exclusively from host
- Relaxing this restriction permits more independence of GPU code from the CPU host, less device/host synchronization required
- Opens possibilities for clearer, less complicated code
- True kernel recursion is now possible
- Allows much more efficient parallelization for some problems
- Without this GPUs lacked a feature which has been available on CPUs for decades, making GPUs harder to program compared to CPUs

Calling kernel inside kernel has same syntax as calling on host

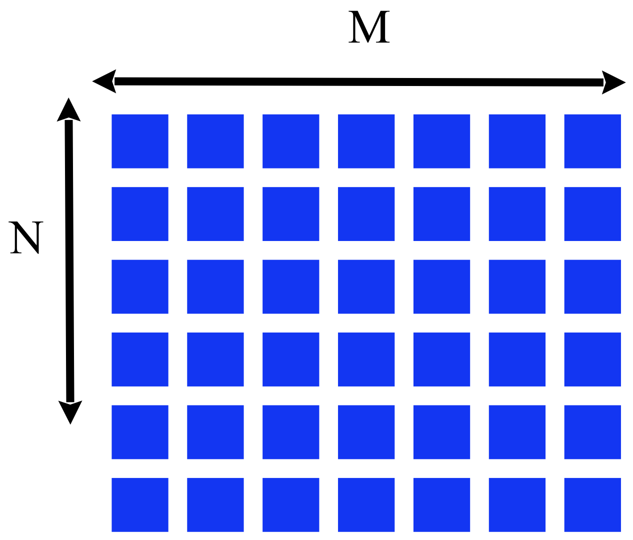
```
__global void Kernel1(){
   ...  //kernel code
   }
__global__ void Kernel2(){
    ...  //kernel code
    Kernel1<<<N,M>>>() // call kernel inside kernel
    ...  //more kernel code
   }
int main(){
   ... //host code
   Kernel2<<K,J>>() // call kernel from host
   ... //more host code
   }
```

- in this example, Kernel2 launches K*J threads when called in the main function
- each thread in Kernel2 that calls Kernel1 will launch N*M new threads
- up to K*J*N*M threads in total can be launched in this case

Program easy to write as an efficient GPU kernel

```
for (int i=0; i<N){
  for (int j=0; j<M;j++){
        some_convolution_function(i,j);
  }
}
```

- Perfect target for 2D thread decomposition
- N*M threads will be launched
- All threads call the same function



Program harder to write as an efficient GPU kernel

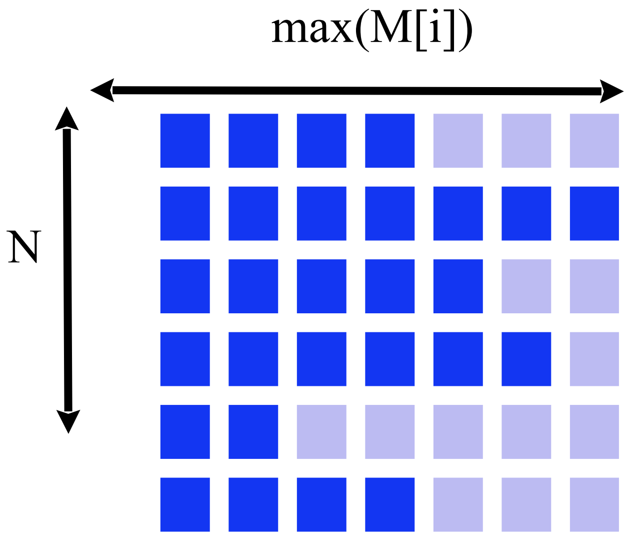
```
for (int i=0; i<N){
  for (int j=0; j<M[i];j++){
        some_convolution_function(i,j);
  }
}
```

- The i,j values no longer fit a regular grid
- Cannot decompose straightforwardly into 2D array of threads
- Can consider a series of bad options

More threads?

```
__global__ oversubscribed_kernel(){
 i=threadIdx.x + blockDim.x * blockIdx.x;
 j=threadIdx.y + blockDim.y * blockIdx.y;
    if(j<M[i]){
           some_convolution_function(i,j);
    }
 }
```

- Could try oversubscription, have N*max(M[i]) threads, some do no work
- This is wasteful, threads which do nothing have to be created



Fewer threads?

```
__global__ void fewer_threads_kernel(){
   i=threadIdx.x + blockDim.x * blockIdx.x;
      for (int j=0; j<M[i];j++){
           some_convolution_function(i,j);
        }
 }
```

- Could have just N threads
- In this case all threads would do work, but N might be too small to create enough threads for efficient use of GPU
- Different threads would likely have very different amounts of work to do
- Warp divergence can become a serious issue

Multiple kernels from host

```
__global__ void multi_kernel(i){

 j=threadIdx.x + blockDim.x * blockIdx.x;
          some_convolution_function(i,j);
 }
//on host
...
for(i=0;i<N;i++){
  multi_kernel<<<(enough threads to handle M[i] evaluations)>>>(i)
}
```

- Could call kernel from host N times
- Individual kernels become much smaller → efficiency can drop dramatically
- Default kernels execute in series, previous must finish before next one starts, even if independent
- Could use streams to avoid this, but number of streams working asynchronously is limited
  - Having a limited number of streams, each running multiple kernels, can be a reasonable solution

New option with CC 3.5: more kernels from GPU

```
__global__ void convolution_kernel(int i,int j){
   some_convolution_function(i,j);
   }
__global__ void dynamic_parallelism_kernel(int *M){

   for(j=0;j<M[blockIdx.x];j++){
           convolution_kernel<<1,1>>>(blockIdx.x,j);
       }
   }
//on host
   ...
   dynamic_parallelism_kernel<<<N,1>>>(M)
```

- First kernel call creates N blocks with 1 thread each, so N threads
- Each thread i of the N threads launches M[i] kernels, each containing one thread
- Another possibility: have each thread i launch a kernel with M[i] threads, but work done by each kernel not equal

New option with CC 3.5: more kernels from GPU

```
__global__ void convolution_kernel(int i,int j){
 some_convolution_function(i,j);
 }
__global__ void dynamic_parallelism_kernel(int *M){
 for(j=0;j<M[blockIdx.x];j++){
          convolution_kernel<<1,1>>>(blockIdx.x,j);
   }
 }
//on host
...
dynamic_parallelism_kernel<<<N,1>>>(M)
```

- Overhead of launching kernels on GPU much lower than on host
- These kernels will execute asynchronously, as they are launched by different threads

Synchronization is an issue, just like on host

```
__global__ void Kernel2(){
...  //kernel code
   Kernel1<<<N,M>>>(); // thread will launch kernel and keep going
   cudaDeviceSynchronize(); // make thread wait for Kernel1 to complete
...  //code that needs data generated by Kernel1
}
```

- Threads will launch kernels and keep on going, just like the host
- If you need for the thread to wait until Kernel1 is finished before continuing, use cudaDeviceSynchronize just like on host
- In this case cudaDeviceSynchronize ensures synchronization only within that 1 GPU thread (!!)

Synchronization is an issue, just like on host

```
__global__ void Kernel2(){
 ...  //kernel code
   if(threadIdx.x==0){
      Kernel1<<<N,M>>>(); // thread will launch kernel and keep going
      cudaDeviceSynchronize(); // make thread wait for Kernel1 to complete
      }
   __syncthreads();  // if all threads in block need Kernel1 to complete
  ...  //code that needs data generated by Kernel1
}
```

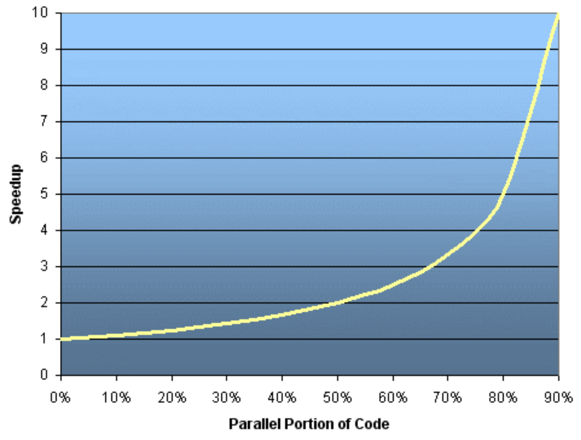
- cudaDeviceSynchronize will not synchronize threads in a block, it only works within thread when inside kernel
- If you want all threads in a block to wait for kernel to finish, use cudaDeviceSynchronize() followed by __syncthreads

Hyper-Q (MPS)

Recall: Amdahl's Law

- Amdahl's Law states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

$$\boxed{\texttt{speedup} = \frac{1}{1 - P}}$$



- If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup). If all of the code is parallelized, P = 1 and the speedup is infinite (in theory).
- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.

Hyper-Q/MPS: why is it important?

- GPUs work well when you saturate them with data-parallel threads.
- P100 GPU has 8 times more cores (so need 8x more threads to get saturated) than Fermi GPU.
- From the Amdahl's law, a code which runs well on Fermi will likely perform poorly on P100.
- Hyper-Q helps to mitigate this, by allowing to share one GPU between different CPU threads.

Exercise: Hyper-Q

- Write a simple CUDA code which has one kernel which uses only one block of threads. (Make the kernel long – say 10 seconds or so; use a serial loop inside for that.)
- This mimics a realistic code which doesn't have enough of parallelism to saturate a modern GPU.
- Time the code without, then with Hyper-Q / MPS. Place the following lines before your code in the job script (or run this manually if using salloc) to use Hyper-Q:

```
mkdir -p $HOME/tmp
export CUDA_MPS_LOG_DIRECTORY=$HOME/tmp
nvidia-cuda-mps-control -d
```

Multi-threaded script HQ.sh

- Use the bash script ~syam/CSE746/MPS/HQ.sh to test MPS

```
#!/bin/bash
for ((i=0; i<$1; i++))
 do
 echo $i
 ./a.out &>out &
 done
wait
```

Profiling CUDA

Profiling

- Profiling is examining code during its execution, to better understand its performance and find ways to improve it
- Essential when developing computationally intensive codes
- As parallel codes are generally computationally intensive, they almost always benefit from profiling
- So far, to develop a fundamental understanding, we have used explicit timing calls, both from CUDA and standard C
- For more complex codes this would become very inconvenient
- Fortunately, CUDA provides a set of powerful profiling tools
- Profiling tools can provide significantly more information than just timings

Profiling tools in CUDA

- Command line profiler: nvprof
  - very fast and convenient
- Nsight IDE
  - has profiler built in
  - diverged into two separate products – ncu-ui for low level profiling of individual kernels, and nsys-ui for whole code profiling (higher level).
- NVIDIA announced that after Volta there will be no support for nvprof. For Amper and newer, only Nsight will work.

Running nsight

- Comes bundled up with “cuda” module. Should use the most recent module available. On graham the following command will load the most recent version of cuda*:

  ```
  $ module load StdEnv/2023 cuda
  ```

- You have to add “-Y” when ssh to the cluster (not needed when using MobaXterm under Windows; for Mac, install Xquartz first).
- In your salloc command, add “--x11” switch.
- Such method (X11 forwarding) can be rather laggy. It is possible (though it’s more complicated) to use VNC method to a compute node on our clusters: https://docs.computecanada.ca/wiki/VNC

nvprof

- Documentation at:http://docs.nvidia.com/cuda/profiler-users-guide/
- Available from command line (after loading a cuda module)
- Will work on standard CUDA executable generated with nvcc, no special flags need to be supplied to nvcc to enable profiling
- Basic usage (here test.x is the executable to be profiled):
  nvprof test.x
- more detailed output listing events on GPU in order of execution
  nvprof --print-gpu-trace   test.x
- more detailed output listing CUDA calls on host
  nvprof --print-api-trace   test.x

Important: compile for right architecture

- It’s especially important when profiling to use the right -arch flag when compiling with nvcc, so it matches the architecture that the executable will be profiled on
- Clearly, since performance optimization is important when profiling, one does not want to introduce any inefficiency by using executable targeted at wrong architecture
- Also, if wrong architecture is selected, some of the profiling diagnostics will be inconsistent

CUDA concepts for today’s lecture

- **Thread occupancy** : getting the most out of the GPU by getting the optimum number of threads actively running on it at any given time
- **Register pressure** : limitations on occupancy and memory efficiency when kernel requires a significant number of registers

Occupancy

- When you execute a CUDA kernel with N threads, generally only a subset of these N threads are resident i.e. have their instructions actively executed at any one time. The remainder have either already finished executing their instructions or are still waiting to begin executing their instructions
- Each multiprocessor inside a CUDA GPU has a hardware limit on the maximum number of threads which can be resident at any one time
- This limit is one of the properties which can be queried at runtime (with *maxThreadsPerMultiProcessor*)

Device diagnostic output

from device_diagnostic.cu in lecture code

output on P100

```
Name:  Tesla P100-PCIE-12GB
Compute capability:  6.0
Clock rate:  1328500
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
   --- Memory Information for device 0 ---
Total global mem:  12790923264
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
   --- MP Information for device 0 ---
Multiprocessor count:  56
Shared mem per mp:  49152
Registers per mp:  65536
Threads in warp:  32
Max threads per multi processor: 2048
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (2147483647, 65535, 65535)
```

Device diagnostic output

from device_diagnostic.cu in lecture code

output on V100

```
Name:  Tesla V100-PCIE-16GB
Compute capability:  7.0
Clock rate:  1380000
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
   --- Memory Information for device 0 ---
Total global mem:  16945512448
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
   --- MP Information for device 0 ---
Multiprocessor count:  80
Shared mem per mp:  49152
Registers per mp:  65536
Threads in warp:  32
Max threads per multi processor: 2048
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (2147483647, 65535, 65535)
```

Device diagnostic output

from device_diagnostic.cu in lecture code

output on A100 (narval)

```
Name:  NVIDIA A100-SXM4-40GB
Compute capability:  8.0
Clock rate:  1410000
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
   --- Memory Information for device 0 ---
Total global mem:  42285268992
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
   --- MP Information for device 0 ---
Multiprocessor count:  108
Shared mem per mp:  49152
Registers per mp:  65536
Threads in warp:  32
Max threads per multi processor: 2048
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (2147483647, 65535, 65535)
```

Occupancy

- The ratio between the actual number of threads that are resident during execution and the theoretical maximum number of threads that could be resident on the GPU
- Broadly speaking, if your occupancy is too low, that indicates inefficiency in your code and should be remedied. Low occupancy may indicate:
  - not all available CUDA cores being used
  - insufficient number of threads for latency hiding (i.e. compensating for slow global memory access by threads)
- On the other hand, **for many codes you should not aim to achieve occupancy close to 1.0**, as code will run sufficiently well even at lower occupancy as it is bound by other limitations

Occupancy - blocks and warps

- It’s also important to note that there is a similar limit on how many **blocks** of threads can be resident at any given time (maxBlocksPerMultiProcessor). This limit is:
  - 32 on P100 / V100 / A100
  - 16 on T4
- Your occupancy will be low if your blocks are too small.
- The maximum number of resident warps is 64 for P100/V100/A100 and 32 for T4.
- It may be more natural to look at warp occupancy i.e. how many warps are resident relative to the maximum

Reasons for occupancy < 1.0

- Blocks are too small, so that (maximum number of possible resident blocks) times (block size) is less than the theoretical limit
- (number of threads in block) does not divide the theoretical limit evenly (eg. block of 1024 threads on GPU with limit 1536)
- Thread blocks consume too much of limited resources (shared memory and registers) so that not enough blocks can be resident to achieve full occupancy
- Note: in all this we assumed for simplicity only one kernel is running at any one time. Multiple concurrent kernels would reduce occupancy per kernel, though possibly enhance it overall.

Estimating occupancy

- Occupancy can be estimated even before you run your code. This can often allow the programmer to pick the optimum grid and block dimensions
- Nvidia used to provide an occupancy calculator in the form of spreadsheet. It is now deprecated (Nsight Compute ncu-ui should be used instead)
- The occupancy calculated this way is the upper limit possible for your code - the actual measured occupancy will be slightly lower, due to overheads associated with launching threads

Limited multiprocessor resources

- The amount of shared memory in a multiprocessor is limited
- If each block of a kernel uses some shared memory, then the total amount used by resident blocks at any one time cannot exceed the total shared memory available
- So, the use of shared memory may reduce the number of resident blocks and hence the number of resident threads, reducing the occupancy
- If number of resident blocks is too low (worst case: just one), then this can put a fundamental limit on occupancy. Thus on P100 this would mean maximum possible occupancy of 0.5, as only 1024 threads (block maximum) can be resident at once if only one block is resident, and that is only half of the theoretical limit

Registers

- Registers constitute the fastest possible memory located right on the multiprocessors. All memory data must go through them.
- Unfortunately, the number of (32-bit) registers on a multiprocessor is limited (64 K on P100, V100, and A100) and they must be shared by all resident threads.
- So the number of registers each individual thread can get is limited to total number divided by number of resident threads. Additionally, there is an upper limit set by Compute Capability (255 on P100/V100/A100).

Register spills

- What happens when the thread-local data is too big to fit in registers allocated to a thread?
- The data will then be “spilled” i.e. moved out from register to local memory
- The “spill” first attempts to store in L1 (24 kB/mp), if that’s not possible, it stores in L2 (4096 kB = 64 kB/mp), and if that’s not possible, it stores in global (device) memory
- When thread tries to get back “spilled” data, it will first try to find it in L1, if it’s not there, it will try L2, and if it’s not there, it will try global memory
- While accesses to L1 are fast, since is located right on the multiprocessor in same space as shared memory, accesses to L2 and global memory are slow

Register usage

- Number of registers per thread is determined by compiler - compiler can output register usage if invoked with

  ```
  nvcc -Xptxas -v
  ```

- Compiler decides how many registers to use. Typically it tries to keep the number used to a minimum, in order to increase occupancy

Register usage

- You can try to force the compiler to use fewer registers with the flag

  ```
  nvcc -maxrregcount=Nmax
  ```

  where Nmax is the maximum number of registers. This is a rather crude way to attempt to optimize as one should usually let the compiler decide, but it can be used for testing
- There is a better way which gives compiler hints on how many registers should be used, by adding to kernel definitions the parameters:

  ```
  __launch_bounds__(maxThreadsPerBlock,minBlocksPerMultiprocessor)
  ```

Register spilling

- The compiler will report how many times it has to spill registers for each thread, and how much local memory it is using
- The number of registers used will also be reported by the profiler
- Be sure you profile with right architecture to get these numbers to match with each other
- But just **knowing the number of register spills is not sufficient to indicate just how bad they are for performance**, as that will depend on whether the spill goes only to L1, or further to L2, or furthest to global memory
- The nature of the spilling can be determined during profiling

Impact of register spilling

- Some small amount of register spilling may not be so bad, especially if spills only go to fast L1 cache
- Problem arises if significant number of spills go to L2 cache. Not only is L2 cache slow, it is need for accesses to global memory, so having it spend a significant amount of time handling spills will impact the effective bandwidth to global memory
- Also, spilling data and then recovering it requires additional instructions in compiled code. More instructions means kernel runs slower. This is only likely to matter in codes which execute a lot of instructions per thread.

Impact of register spilling on L2 cache

- nvprof profiler can provide very specific counts for many events. To see all available, execute:

```
nvprof --query-events
```

- To count events describing L1 cache access, you can run:

```
nvprof –events
l1_local_load_hit,l1_local_load_miss,l1_local_store_m
iss,l1_local_store_hit
```

- The ratio of hits and misses indicates how often data is located in fast L1 cache, and how often data is not found in L1 cache (and so has to be looked for in slow L2 cache)
- If you have lots of misses relative to hits, that is bad for performance

Reducing impact of spilling to L2

- Try increasing register count at compile stage (occupancy decrease)
- Can try non-caching loads, with -Xptxas -dlcm=cg . This means global memory accesses will avoid using L1 cache. The tradeoff is that access to global memory is slower, but there is more L1 cache to handle register spilling.
- (as an aside, non-caching load brings in data in 32-byte chunks, caching in 128-byte chunks. Sometimes one is better than other.)

The end