CUDA Programming Introduction III

Nikos Hardavellas

Some slides/material from:
UToronto course by Andreas Moshovos
UIUC course by Wen-Mei Hwu and David Kirk
UCSB course by Andrea Di Blas
Universitat Jena by Waqar Saleem
NVIDIA by Simon Green and many others

CUDA Limits (capability 3.0)

- Grid and block dimension restrictions
  - Grid: 2G x 64K x 64K
  - Block: 1024 x 1024 x 64
  - Max threads/block = 1024
- A block maps onto an SM
  - Up to 16 blocks per SM, 64 warps, 2K threads
- Every thread uses registers
  - Up to 63 regs per thread, 64K per SM
- Every block uses shared memory
  - Up to 48KB shared memory (or 32K or 16K, user configured)
- Example:
  - 16x16 blocks of threads using 20 regs each
  - Each block uses 8K of shared memory
    - 5120 registers / block → **⌊12.8⌋ blocks/SM**
    - 8K shared memory/block → 6 blocks/SM
    - Only 6 blocks will be scheduled per SM; half the registers will stay unused!

Predefined Vector Datatypes

- Can be used both in host and in device code.

```
[u]char[1..4]
[u]short[1..4]
[u]int[1..4]
[u]long[1..4]
float[1..4]
```

- Structures accessed with `.x, .y, .z, .w` fields
- Default constructors, `make_TYPE(…)`

```
float4 f4 = make_float4 (1f, 10f, 1.2f, 0.5f);
```

- **`dim3`**
  - type built on `uint3`
  - Used to specify dimensions
  - Default value is (1, 1, 1)

Error Handling

- Most `cuda…()` functions return a `cudaError_t`
  - If `cudaSuccess`: Request completed without a problem

- `cudaGetLastError():`
  - Returns the last error to the CPU
  - Use with `cudaThreadSynchronize()`

```
cudaError_t code;
cudaThreadSynchronize ();
code = cudaGetLastError ();
```

- `char *cudaGetErrorString(cudaError_t code);`
  - Returns a human-readable description of the error code

Error Handling Utility Function

```
void
cudaDie (const char *msg)
{
  cudaError_t err;
  cudaThreadSynchronize ();
  err = cudaGetLastError();

  if (err == cudaSuccess) return;
  fprintf (stderr,
             "CUDA error: %s: %s.\n",
             msg,
             cudaGetErrorString (err));
  exit(EXIT_FAILURE);
}
```

Adapted from:
http://www.ddj.com/hpc-high-performance-computing/207603131

Error Handling Macros

- **CUDA_SAFE_CALL (some cuda call)**

```
CUDA_SAFE_CALL (cudaMemcpy( a_h,
                            a_d,
                            arr_size,
                            cudaMemcpyDeviceToHost)
               );
```

- Prints error and exits on error
- Must define `#define _DEBUG`
  - No checking code emitted when undefined: higher performance
- Use `make dbg=1` under `NVIDIA_CUDA_SDK`

Measurement Methodology

- You will not get exactly the same time measurements every time
  - Other processes running
  - External events (e.g., network activity)
  - Cannot control all system aspects
  - “Non-determinism”
- Must take sufficient samples
  - Say 10 or more
  - There is theory on what the number of samples must be
- Report average, excluding highest and lowest outliers

Measuring Time – `gettimeofday()`

Unix-based:

```
#include <sys/time.h>
#include <time.h>
struct timeval start, end;
...
gettimeofday (&start, NULL);
   ...computation we are interested in...
gettimeofday (&end, NULL);

timeCpu = (float)(end.tv_sec - start.tv_sec);
if (end.tv_usec < start.tv_usec) {
  timeCpu -= 1.0;
  timeCpu += (double)
      (1000000.0 + end.tv_usec - start.tv_usec)/1000000.0;
} else {
  timeCpu += (double)
             (end.tv_usec - start.tv_usec)/1000000.0;
}
```

Measuring Time – Using CUDA `clock()`

- `clock_t clock ();`
- Can be used in device code
- Returns a counter value
  - One per multiprocessor. Incremented on every clock cycle
- Sample at the beginning and end of the code
- Upper bound since threads are time-sliced

```
uint start = clock();
   ... compute (less than 3 sec) ....
uint end = clock();
if (end > start)
      time = end - start;
else
      time = end + (0xffffffff - start)
```

- Look at the clock example under projects in SDK
- Using takes some effort
  - Every thread measures start and end
  - Then must find min start and max end (global across threads)
  - Cycle accurate

Measuring Time – Using `cutTimer...()` library

```
#include <cuda.h>
#include <cutil.h>
unsigned int htimer;
...
cutCreateTimer (&htimer);
...
CudaThreadSynchronize ();
cutStartTimer(htimer);
   ...computation we are interested in...
cudaThreadSynchronize ();
cutStopTimer(htimer);

printf ("time: %f\n", cutGetTimerValue(htimer));
```

Code Overview: Host side

```
#include <cuda.h>
#include <cutil.h>
unsigned int htimer;
float *ha, *da;
main (int argc, char *argv[]) {
  int N = atoi (argv[1]);
  ha = (float *) malloc (sizeof (float) * N);
  for (int i = 0; i < N; i++)
      ha[i] = i;
  cutCreateTimer(&htimer);
  cudaMalloc((void **) &da, sizeof (float) * N);
  cudaMemCpy((void *)da, (void *)ha, sizeof(float)*N, cudaMemcpyHostToDevice);
  blocks = (N + threads_block – 1) / threads_block;
  cudaThreadSynchronize();
  cutStartTimer(htimer);
  darradd <<<blocks, threads_block>> (da, 10f, N)
  cudaThreadSynchronize();
  cutStopTimer(htimer);
  cudaMemCpy((void *)ha, (void *)da, sizeof(float)*N, cudaMemcpyDeviceToHost);
  cudaFree(da);
  free(ha);
  printf(“processing time: %f\n", cutGetTimerValue(htimer));
}
```

Code Overview: Device Side

```
__device__ float addmany (float a, float b, int count)
{
   while (count--) a += b;
   return a;
}



__global__ darradd (float *da, float x, int N)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) da[i] = addmany (da[i], x, 10);
}
```

Handling Large Input Data Sets – 1D Example

- Recall CUDA 5.0 limits: gridDim.x ≤ 2147483647
  - In CUDA 4.0: gridDim.x ≤ 65535
- Data set may be too big to fit optimally in a single grid
- Host may call kernel multiple times:

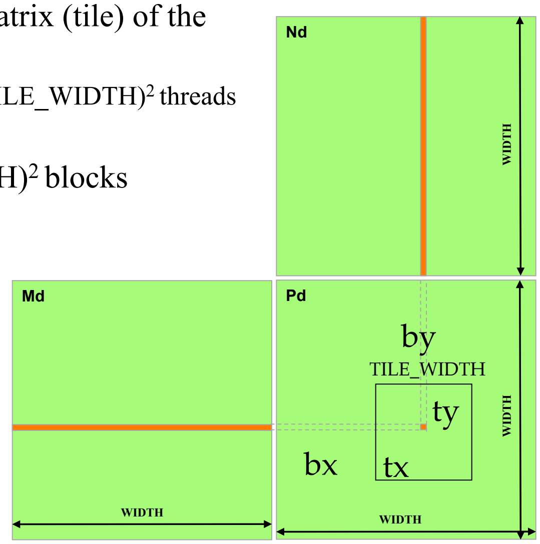
```
float *dac = da; // starting offset for current kernel
while (n_blocks)
{
    int bn = n_blocks < 2147483647 ? n_blocks : 2147483647;
    int elems;  // array elements processed in this kernel
    elems = bn * block_size;
    darradd <<<bn, block_size>>> (dac, 10.0f, elems);

    n_blocks -= bn;
    dac += elems;
}
```

- Potentially better alternative:
  - Each thread processes multiple elements

Handling Arbitrary-Size Square Matrices (example)

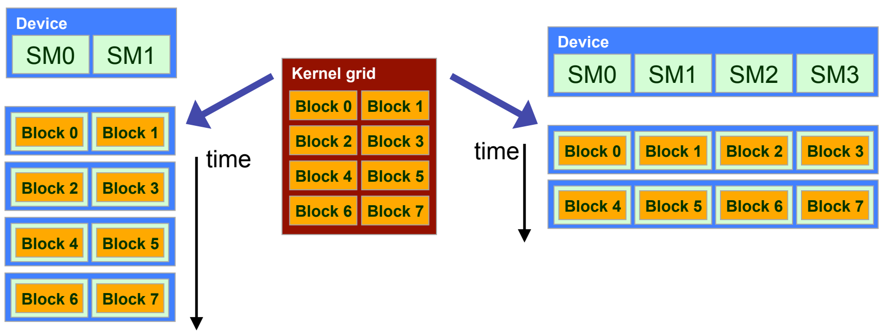
- Have each 2D thread block compute a $(\text{TILE\_WIDTH})^2$ sub-matrix (tile) of the result matrix
  - Each thread block has $(\text{TILE\_WIDTH})^2$ threads
- Generate a 2D grid of $(\text{WIDTH / TILE\_WIDTH})^2$ blocks

You still need to put a loop around the kernel call for cases where WIDTH / TILE_WIDTH is greater than max grid size



Transparent Scalability

- CUDA makes no guarantees on thread block execution order
  - Each thread block can execute in any order relative to other blocks
- Hardware is free to assign thread blocks to any SM at any time
  - Program correctness is preserved across different architectures
  - A kernel can run on a GPU with 2 SMs, or on a GPU with 4 SMs, or ...
- Similarly for threads
- **Transparent scalability**: kernel scales to any number of cores
  - Up to the max number of blocks and threads in the computation grid



Floating Point Considerations

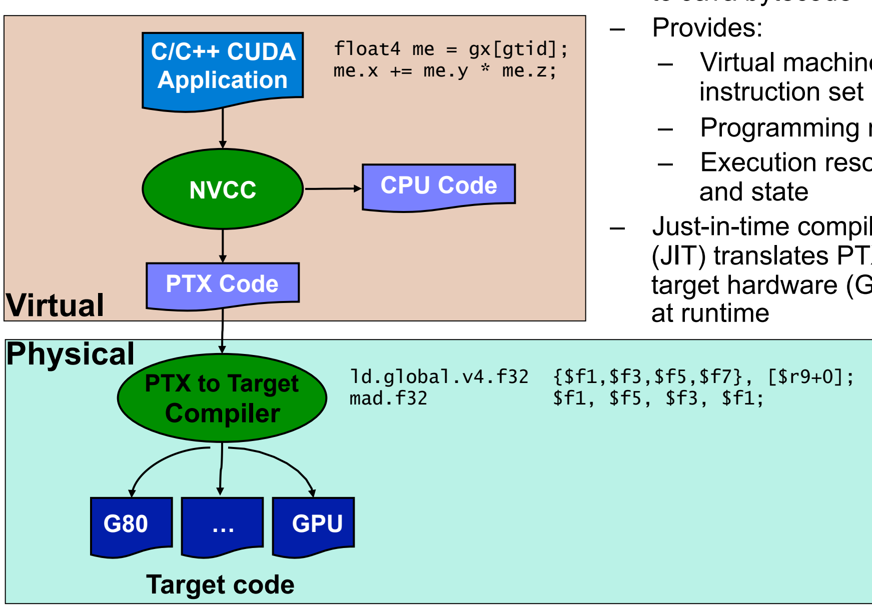
- **Floating-point computations on GPU may slightly differ from CPU**
  - Different FP implementation, instruction sets
  - Use of extended precision for intermediate results on CPU
    - Can compare results only within some margin of error
    - Lab scaffold code takes that into account
- Performance
  - If double precision is not needed, do not use it
  - Mixed-precision methods

Some Useful Information on Tools

- Please look at user guides for more info
- CUDA documentation on blackboard

Compiling a CUDA Program

Parallel Thread eXecution (PTX)



- PTX for GPUs is similar to Java bytecode
- Provides:
  - Virtual machine and instruction set
  - Programming model
  - Execution resources and state
- Just-in-time compiler (JIT) translates PTX to target hardware (GPU) at runtime

Compilation

- Any source file containing CUDA language extensions must be compiled with NVCC
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC outputs:
  - C code (host CPU Code)
    - Must then be compiled with the rest of the application using another tool
  - PTX
    - Object code directly
    - Or, PTX source, interpreted at runtime

Linking

- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (`cudart`)
  - The CUDA core library (`cuda`)

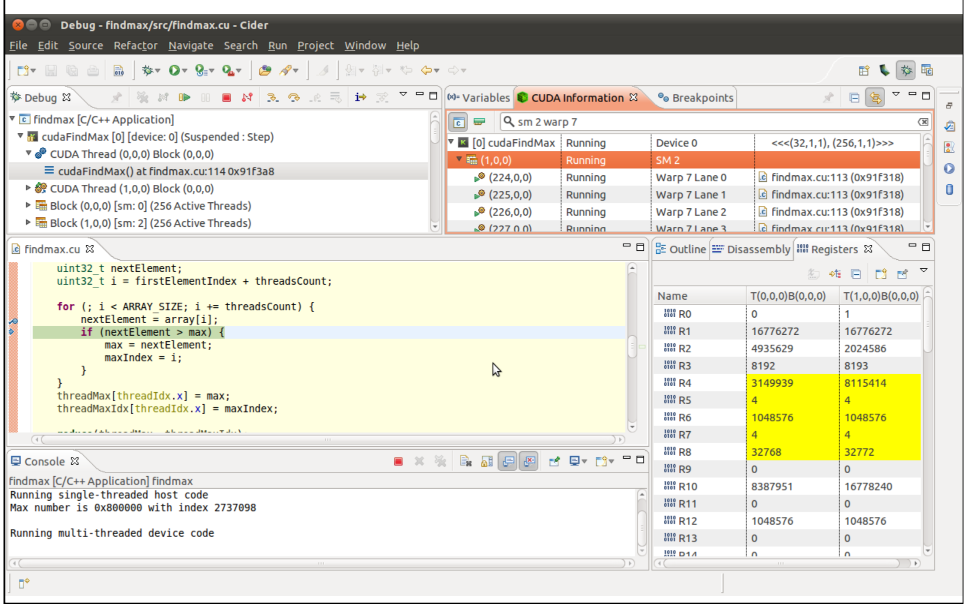
Debugging and Profiling

- **Nsight Debugger**
  - Seamless and simultaneous debugging of both CPU and GPU code
  - View program variables across several CUDA threads
  - Examine execution state and mapping of the kernels and GPUs
  - View, Navigate and filter to selectively track execution across threads
  - Set breakpoints and single-step execution at source-code or assembly
  - Includes cuda-memcheck to help detect memory errors
  - Debugging on the GPU rather than SW emulation
- **Nsight Profiler**
  - Easily identify performance bottlenecks using a unified CPU and GPU trace of application activity
  - Automated analysis system pin-points optimization opportunities
  - Highlights potential performance problems at specific source-lines within application kernels
  - Close integration with Nsight Editor and Builder enable fast edit-build-profile optimization cycle

https://developer.nvidia.com/nsight-eclipse-edition

Debugging and Profiling

Debugger:



Debugging and Profiling

Profiler: