Lecture 5

An improved matrix multiply

Occupancy

Thread divergence
Today’s lecture

• Occupancy and latency
• Further Improvements to Matrix Multiply
• Thread divergence
Projects

• Counts for 65% of your grade
• Complete in 7 weeks
  ♦ Proposal: in class presentation + writeup (1/27)
  ♦ Progress report #1: presentation (2/10)
  ♦ Progress report #2: writeup (2/26)
  ♦ Final Presentation: 3/13*
  ♦ Final Report: 3/16 (Monday) at 5pm
Occupancy

• A minimum number of warps needed to hide memory latency
• **Occupancy:** \# active warps ÷ max \# warps supported by vector unit
• Limited by vector unit resources
  - Amount of shared memory
  - Number of registers
  - Maximum number of threads
• Consider a kernel (16x16 block size)
  - Shared memory/block = 4096 bytes
  - Reg/thread=30
    2048 max threads/SMX = 8 blocks
    30reg/blk×256thrds/blk×8blks = 61440 < 64k
  - # available regs not limiting factor are close
• Tradeoff: more blocks with fewer threads or more threads with fewer blocks
  - Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
  - Register consumption
• Maximizing the occupancy may not maximize performance
### Occupancy calculation with 16 x 16 threads

\[
\text{Occupancy} = \frac{\# \text{ active warps per SM}}{\text{Maximum possible \# active warps}}
\]

Physical Limits for GPU Compute Capability: 2.0

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads per Warp</td>
<td>32</td>
</tr>
<tr>
<td>Warps per Multiprocessor</td>
<td>48</td>
</tr>
<tr>
<td>Threads per Multiprocessor</td>
<td>1536</td>
</tr>
<tr>
<td>Thread Blocks per Multiprocessor</td>
<td>8</td>
</tr>
<tr>
<td>Total # of 32-bit registers per Multiprocessor</td>
<td>32768</td>
</tr>
<tr>
<td>Register allocation unit size</td>
<td>64</td>
</tr>
<tr>
<td>Register allocation granularity</td>
<td>warp</td>
</tr>
<tr>
<td>Registers per Thread</td>
<td>63</td>
</tr>
<tr>
<td>Shared Memory per Multiprocessor (bytes)</td>
<td>49152</td>
</tr>
<tr>
<td>Shared Memory Allocation unit size</td>
<td>128</td>
</tr>
<tr>
<td>Warp allocation granularity</td>
<td>2</td>
</tr>
<tr>
<td>Maximum Thread Block Size</td>
<td>1024</td>
</tr>
</tbody>
</table>

#### CUDA GPU Occupancy Calculator

1.) Select Compute Capability (click): **2.0**
1.b) Select Shared Memory Size Config (bytes) **49152**

2.) Enter your resource usage:
   - Threads Per Block: **256**
   - Registers Per Thread: **25**
   - Shared Memory Per Block (bytes): **0**

(Do not edit anything below this line)

3.) GPU Occupancy Data
   - Active Threads per Multiprocessor: **1024**
   - Active Warps per Multiprocessor: **32**
   - Active Thread Blocks per Multiprocessor: **4**
   - Occupancy of each Multiprocessor: **67%**

Allocated Resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Per Block</th>
<th>Limit Per SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warps</td>
<td>8</td>
<td>48</td>
</tr>
<tr>
<td>Registers (Warp limit per SM due to per-warp reg count)</td>
<td>8</td>
<td>38</td>
</tr>
<tr>
<td>Shared Memory (Bytes)</td>
<td>0</td>
<td>49152</td>
</tr>
</tbody>
</table>

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor

<table>
<thead>
<tr>
<th>Limitation</th>
<th>Blocks/SM</th>
<th>* Warps/Block = Warps/SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited by Max Warps or Max Blocks per Multiprocessor</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Limited by Registers per Multiprocessor</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Limited by Shared Memory per Multiprocessor</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Physical Max Warps/SM = 48
Occupancy = 32 / 48 = 67%

Scott B. Baden / CSE 262 / UCSD, Wi '15
Full occupancy
Latency

- Instructions wait on dependencies
  \[ x = a + b; \quad \text{// } \sim 20 \text{ for floating point, } 500+ \text{ for memory} \]
  \[ y = a + c; \quad \text{// independent (stall)} \]
  \[ z = x + d; \quad \text{// dependent, wait} \]

- How many warps are needed to hide latency (100% throughput)
  if minimum latency is 4 cycles / instruction?

\[
\text{# Parallelism (threads)} = \text{latency} \times \text{throughput} \quad T = \lambda \times p
\]

480 mul+adds/cycle; 32 memory ops /cycle [1.3 device]

- Required parallelism depends on op; for single precision
  - GT200 (C1060): 24 CP * 8 cores / SM = 192 ops/SM
  - GF104 (GTX 460, Cseclass03-07): 18 CP * 48 = 864
  - GK110?
Thread vs instruction level parallelism

- We are told to maximize the number of threads
- But we can also use instruction level parallelism to boost performance at a lower occupancy
  - See http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf
- On GT200, 100% peak with 25% occupancy
  \[ 192 \text{ ops / cycle} = 8 \text{ warps / 32 max possible warps} \]
- On the GF104, we need ILP to go beyond 66% of peak
  - 48 cores/SM, half warp (16 cores) issues at a time
  - But we have only 2 schedulers
  - We must issue 2 independent instructions per warp in the same cycle
- GK110?

576 threads needed for 100% utilization

```c
#pragma unroll UNROLL
for( i = 0; i < N_Iter; i++ ){
    a = a * b + c;
}
```

320 threads needed for 100% utilization

```c
#pragma unroll UNROLL
for( i = 0; i < N_Iter; i++ ){
    a = a * b+c;
    d = d * b + c;
}
```
Hiding memory latency

- **Parallelism** = latency \( \times \) throughput

  Arithmetic: \(576 \text{ ops/SM} = 18 \text{CP} \times 32/\text{SM/CP}\)
  Memory: \(150\text{KB} \approx 500\text{CP} \times 1100\text{nsec} \times 150\text{GB/sec}\)

- **How can we keep 150KB in flight?**
  - Multiple threads: \(~35,000\) threads \(\times\) 4B/thread
  - ILP (increase fetches per thread)
  - Larger fetches (64 or 128 bit/thread)
  - Higher occupancy

Copy 1 float /thread, need 100% occupancy

```c
int indx = threadIdx.x + block * blockDim.x;
float a0 = src[indx];
dest[indx] = a0;
```

Copy 2 floats /thread, need 50% occ

```c
float a0 = src[indx];
float a1 = src[indx+blockDim.x];
dest[indx] = a0;
dst[index+blockDim.x] = a1;
```

Copy 4 floats /thread, need 25% occ

```c
int indx = threadIdx.x + 4 * block * blockDim.x;
float a[4]; // in registers
for(i=0; i<4; i++) a[i] = src[indx+i*blockDim.x];
for(i=0; i<4; i++) dst[indx+i*blockDim.x]=a[i];
```
Today’s lecture

• Occupancy and latency
• Further Improvements to Matrix Multiply
• Thread divergence
How to improve matrix multiply still further

• Volkov and Demmel, SC08
• Hide arithmetic latency using fewer threads
• Hide memory latency using fewer threads
• Improving performance using fewer threads
  • We can reduce number of threads through lower occupancy …
  • ..by making better use of registers we can trade locality against parallelism
• Code was implemented on a 1.x device, so some details will be different (more registers on Kepler, for example)
Incremental improvements to matrix multiply

- Follows V. Volkov [GTC10]
- Gets 137 Gflops / sec

```c
float Csub = 0;
for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    __syncthreads();

    #pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += AS(ty, k) * BS(k, tx);
    __syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```
Two outputs / thread

- 2 outputs, double the loads

```cpp
float Csub[2] = {0, 0}; // array is allocated in registers
for (int a = aBegin, b = bBegin; a <= aEnd;
     a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    __syncthreads();
```
Two outputs / thread, part 2

• ×2 flops and stores
• 341 Gflops/sec

```c
#pragma unroll
for (int k = 0; k < BLOCK_SIZE; ++k)
{
    Csub[0] += AS(ty, k) * BS(k, tx);
    Csub[1] += AS(ty+16, k) * BS(k, tx);
}
__syncthreads();
```

```c
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+16) + tx] = Csub[1];
```
4 outputs / thread

```c
float Csub[4] = {0,0,0,0}; // array is in registers
for (int a = aBegin, b = bBegin; a <= aEnd;
     a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+8, tx) = A[a + wA * (ty+8) + tx];
    BS(ty+8, tx) = B[b + wB * (ty+8) + tx];
    AS(ty+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    AS(ty+24, tx) = A[a + wA * (ty+24) + tx];
    BS(ty+24, tx) = B[b + wB * (ty+24) + tx];
    __syncthreads();
```
4 outputs / thread

- 427 Gflops/sec [w/8 output/thread → 485 Gflops/s)
- ×2 # registers
- 50% occupancy

```c
#pragma unroll
for (int k = 0; k < BLOCK_SIZE; ++k)
{
    Csub[0] += AS(ty, k) * BS(k, tx);
    Csub[1] += AS(ty+8, k) * BS(k, tx);
    Csub[2] += AS(ty+16, k) * BS(k, tx);
    Csub[3] += AS(ty+24, k) * BS(k, tx);
}
__syncthreads();
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+8) + tx] = Csub[1];
C[c + wB * (ty+16) + tx] = Csub[2];
C[c + wB * (ty+24) + tx] = Csub[3];
```
Volkov and Demmel’s SGEMM

- Improve performance using fewer threads
  - Reducing concurrency frees up registers to trade locality against parallelism
  - ILP to increase processor utilization

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
\[ c[1:16] = 0 \]
\[ \text{do} \]
\[ b[1:16][1:16] = \text{next } 16 \times 16 \text{ block in } B \text{ or } B^T \]
\[ \text{local barrier} //\text{wait until } b[i][j] \text{ is written by all warps} \]
\[ \text{unroll for } i = 1 \text{ to } 16 \text{ do} \]
\[ a = \text{next } 64 \times 1 \text{ column of } A \]
\[ c[1] := a^T b[i][1] //\text{rank-1 update of } C \text{’s block} \]
\[ c[2] := a^T b[i][2] //\text{data parallelism } = 1024 \]
\[ c[3] := a^T b[i][3] //\text{stripmined in software} \]
\[ \ldots \]
\[ c[16] := a^T b[i][16] //\text{access to } b[i][j] \text{ is stride-1} \]
\[ \text{endfor} \]
\[ \text{local barrier} //\text{wait until done using } b[i][j] \]
update pointers in A and B
\[ \text{repeat until} \text{ pointer in } B \text{ is out of range} \]
Merge c[1:16] with 64×16 block of C in memory
SGEMM Code

```c
__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 )
        {
        }
    __syncthreads();
    } while( B < Blast );
    for( int i = 0; i < 16; i++, C += ldc )
        C[0] = alpha*c[i] + beta*C[0];
}
```

Volkov and Demmel

Scott B. Baden / CSE 262 / UCSD, Wi '15
Data motion cost

- Communication performance is a major factor in determining the overall performance of an application

- The $\alpha-\beta$ model: $\alpha + \beta^{-1} n$
  
  $n = \text{message length}$
  
  $\alpha = \text{message startup time}$
  
  $\beta_{\infty} = \text{peak bandwidth (bytes / second)}$

<table>
<thead>
<tr>
<th>Machine</th>
<th>$\beta_{\infty}$ (Dev)</th>
<th>H-D</th>
<th>D-H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stampede</td>
<td>144 GB/s</td>
<td>5.9</td>
<td>6.4</td>
</tr>
<tr>
<td>CseClass04</td>
<td>56.1</td>
<td>5.2</td>
<td>4.1</td>
</tr>
</tbody>
</table>

As reported by bandwidthTest

Scott B. Baden / CSE 262 / UCSD, Wi '15
Half power point

- We define the **half power point** \( n_{1/2} \) as the transfer size required to achieve \( \frac{1}{2} \beta_\infty \)
  \[
  \frac{1}{2} \beta^{-1}_\infty = \frac{n_{1/2}}{T(n_{1/2})} \Rightarrow \beta^{-1}(n_{1/2}) = \frac{1}{2} \beta^{-1}_\infty
  
  \]

- In theory, this occurs when \( \alpha = \beta^{-1}_\infty n_{1/2} \Rightarrow n_{1/2} = \alpha \beta_\infty \)

- Formula may not be accurate

**(SDSC Blue Horizon)**

Scott B. Baden / CSE 262 / UCSD, Wi '15
Consequences of data motion cost

• Consider saxpy: \( z[i] = a \cdot x[i] + y[i] \)
  Performs 2 flops per 3 words of memory written and read

• This is a bandwidth bound kernel
  Running time \( \approx \alpha + (\text{Required Bandwidth}) \cdot \beta^{-1}_\infty \)
  \( \alpha = 4 \mu s, \quad \beta_\infty = 127 \text{ GB/sec} \)

• Flop rate bounded by
  \((2n \text{ flops} / 12n \text{ bytes}) \times 127 \text{ GB/sec} = 27 \text{ Gflops/sec}\)

• \( N_{1/2} \) Half bandwidth point: \( N \approx 42,000 \)
  Half the time is spent in the \( \alpha \) term: 8\( \mu \)s

• Saxpy is used in matrix linear algebra, with matrices
  ♦ But matrix size \( N \) cannot exceed \( \sqrt{M/24} \)
    \( M \sim 1 \text{ GB} \Rightarrow N \approx 16,000 \)

♦ Consequence: saxpy takes constant time to run for practical matrix sizes
Today’s lecture

• Projects
• Further Improvements to Matrix Multiply
• Thread divergence
Thread Divergence

- All the threads in a warp execute the same instruction
- Different control paths are serialized
- *Divergence when a predicate* is a function of the thread Id
  
  \[
  \text{if (threadId < 2) \{ \}}
  \]

- No divergence if all follow the same path
  
  \[
  \text{if (threadId / WARP\_SIZE < 2) \{ \}}
  \]

- Consider reduction, e.g. summation \( \sum_i x_i \)
Thread divergence

• All the threads in a warp execute the same instruction
• Different control paths are serialized
Divergence example

if (threadIdx >= 2)
    a=100;
else
    a=-100;

compare threadIdx,2
Divergence example

if (threadIdx >= 2) 
   a=100;
else
   a=-100;

Mary Hall

Scott B. Baden / CSE 262 / UCSD, Wi '15
Divergence example

if (threadIdx >= 2)
    a=100;
else
    a=-100;

Mary Hall

Scott B. Baden / CSE 262 / UCSD, Wi '15
Thread Divergence

• All the threads in a warp execute the same instruction
• Different control paths are serialized
• *Divergence* when a predicate is a function of the threadId
  
  \[
  \text{if (threadId} < 2) \{ \}\]

• No divergence if all follow the same path within a warp
  
  \[
  \text{if (threadId} / \text{WARP}_\text{SIZE} < 2) \{ \}\]

• We can have different control paths within the thread block
Example – reduction – thread divergence

Thread 0  Thread 2  Thread 4  Thread 6  Thread 8

0  1  2  3  4  5  6  7  8  9  10  11

1  0+1  2+3  4+5  6+7  8+9  10+11

2  0..3  4..7  8..11

3  0..7  8..15

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Scott B. Baden / CSE 262 / UCSD, Wi '15
The naïve code

```c
__global__ void reduce(int *input, unsigned int N, int *total){
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    __shared__ int x[BSIZE];
    x[tid] = (i<N) ? input[i] : 0;
    __syncthreads();

    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        __syncthreads();
        if (tid % (2*stride) == 0)
            x[tid] += x[tid + stride];
    }
    if (tid == 0) atomicAdd(total,x[tid]);
}
```

Scott B. Baden / CSE 262 / UCSD, Wi '15
Reducing divergence and avoiding bank conflicts

Thread 0
The improved code

- All threads in a warp execute the same instruction
- No divergence until stride < 32
- All warps active when stride $\geq 32$

```c
__shared__ int x[ ];
unsigned int tid = threadIdx.x;
unsigned int s;

for (s = blockDim.x/2;
    s > 1;
    s /= 2) {
    __syncthreads();
    if (tid < s )
        x[tid] += x[tid + s ];
}
```

```c
for (stride = 1;
    stride < blockDim.x;
    stride *= 2) {
    __syncthreads();
    if (tid % (2*stride) == 0)
        x[tid] += x[tid + stride];
}
```

- All threads in a warp execute the same instruction
- reduceSum $<<<N/512,512>>> (x,N)$
- No divergence until stride < 32
- All warps active when stride $\geq 32$

<table>
<thead>
<tr>
<th>$s$</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>0:255</td>
</tr>
<tr>
<td>128</td>
<td>0:127</td>
</tr>
<tr>
<td>64</td>
<td>0:63</td>
</tr>
<tr>
<td>32</td>
<td>0:31</td>
</tr>
<tr>
<td>16</td>
<td>0:15</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Summary - Programming issues

• Branches serialize execution within a warp
• Tradeoff: more blocks with fewer threads or more threads with fewer blocks
  ♦ Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
  ♦ Register consumption
  ♦ Scheduling: hide latency
• Shared memory and registers do not persist across kernel invocations
Recapping: Maximize Performance on a GPU

- Avoid algorithms that present intrinsic barriers to utilizing the hardware
  - Avoid costly branches, or render harmless
  - Minimize serial sections
- Cut data motion costs
  - Hide latency of host ↔ device memory transfers
  - Reduce global memory accesses → fast on-chip accesses
  - Coalesced memory transfers
- Improve parallelism
  - Vectorization, SSE
  - Pipelining
  - Instruction level parallelism
Branch Predication

- All instructions support predication
- Condition code or *predicate* per thread: set to *true* or *false*
- Execute only if the predicate is true

```c
if (x>1) {
    y = 7;
    test = (x>1)
    test: y=7
}
```

- Compiler replaces a branch instruction with predicated instructions only if the number of instructions controlled by branch condition is not too large
- If the compiler predicts too many divergent warps...
  threshold = 7, else 4
Concurrency – Host & Device

• Nonbocking operations
  • Asynchronous Device↔ {Host,Device}
  • Kernel launch
• Interleaved CPU and device calls: kernel calls run asynchronous with respect to host
• But a kernel call sequence runs sequentially on device
• Multiple kernel invocations running in separate CUDA streams: interleaved, independent computations

  cudaMemcpyAsync(d1, h1, N, H2D, st1);
  cudaMemcpyAsync(d2, h1, N, H2D, st2);
  cudaMemcpyAsync(h4, d4, N, D2H, st3);
  cudaMemcpyAsync(d3, h4, N, D2H, st4);
  cudaMemcpyAsync(d4, h4, N, D2H, st4);
  cudaMemcpyAsync(d1, h1, N, H2D, st1);
  cudaMemcpyAsync(d2, h1, N, H2D, st2);
  cudaMemcpyAsync(h4, d4, N, D2H, st3);
  cudaMemcpyAsync(d3, h4, N, D2H, st4);
  cudaMemcpyAsync(d4, h4, N, D2H, st4);
  cudaMemcpyAsync(d1, h1, N, H2D, st1);
  cudaMemcpyAsync(d2, h1, N, H2D, st2);
  cudaMemcpyAsync(h4, d4, N, D2H, st3);
  cudaMemcpyAsync(d3, h4, N, D2H, st4);
  cudaMemcpyAsync(d4, h4, N, D2H, st4);
  cudaMemcpyAsync(d1, h1, N, H2D, st1);
  cudaMemcpyAsync(d2, h1, N, H2D, st2);
  cudaMemcpyAsync(h4, d4, N, D2H, st3);
  cudaMemcpyAsync(d3, h4, N, D2H, st4);
  cudaMemcpyAsync(d4, h4, N, D2H, st4);
  cudaMemcpyAsync(d1, h1, N, H2D, st1);
  cudaMemcpyAsync(d2, h1, N, H2D, st2);
  cudaMemcpyAsync(h4, d4, N, D2H, st3);
  cudaMemcpyAsync(d3, h4, N, D2H, st4);
  cudaMemcpyAsync(d4, h4, N, D2H, st4);

on-demand.gputechconf.com/gtc-express/2011/presentations/
StreamsAndConcurrencyWebinar.pdf
Measuring performance

• Two ways
  ❖ Use an ordinary timer, e.g. gettimeofday()
  ❖ Use Cuda events/elapsed time (#ifdef CUDA_TIMER)

• See incrArray

• Note that kernel invocation is asynchronous

```c
cudaThreadSynchronize();
double t_device_compute = -getTime();
    incr<<< nBlocks, bSize >>> (a_d, N);
cudaThreadSynchronize();
t_device_compute +=getTime();
```

• Nvprof profiler
CUDA Error Handling

Cuda error: Can't run kernel: invalid device function.

• Cuda can silently fail, you can observe misleading performance
• E.g. if you specify an invalid grid / thread block dimensions
• Note: the last error can be cleared by successive kernel calls, so check frequently

```
cudamalloc((void **) &a_d, size);
checkCUDAError("Unable to allocate storage on the device");
```

• Consult checkCUDAError() in utils.cu (incrArr)
• What about asynchronous calls?
• cf CUDA Programming Guide, “Error Handling”
Getting information about the binary

- Compiler will report a kernel’s register usage along with that of local, shared and constant memory
  --ptxas-options=-v

incrementArrays (float *a, int N)
int idx = blockIdx.x*blockDim.x + threadIdx.x;
if (idx<N) a[idx] = a[idx]+1.f;

ptxas info : Compiling entry function
'_Z22incrementArrayOnDevicePfii' for 'sm_21'
ptxas info : Used 6 registers, 48 bytes cmem[0]
Hardware evolution

- Kepler
- Latest entry is Maxwell