Lecture 10
CSE 260 – Parallel Computation
(Fall 2015)
Scott B. Baden

Looking at PTX code
Thread Scheduling
Announcements

• Weds office hours moved to 2:00 to 3:30 this week only (10/28)

• Next Weds office hours will end at 3:30 instead of 4pm (11/4)
<table>
<thead>
<tr>
<th></th>
<th>Fall 2015</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Peak</strong></td>
<td>9.32</td>
</tr>
<tr>
<td>ATLAS</td>
<td>6.78</td>
</tr>
<tr>
<td></td>
<td>5.25</td>
</tr>
<tr>
<td></td>
<td>5.04</td>
</tr>
<tr>
<td></td>
<td>4.9</td>
</tr>
<tr>
<td></td>
<td>4.44</td>
</tr>
<tr>
<td></td>
<td>3.83</td>
</tr>
<tr>
<td></td>
<td>3.82</td>
</tr>
<tr>
<td></td>
<td>3.76</td>
</tr>
<tr>
<td></td>
<td>3.75</td>
</tr>
<tr>
<td></td>
<td>3.72</td>
</tr>
<tr>
<td></td>
<td>3.52</td>
</tr>
<tr>
<td></td>
<td>3.41</td>
</tr>
<tr>
<td></td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>3.28</td>
</tr>
<tr>
<td></td>
<td>3.21</td>
</tr>
<tr>
<td></td>
<td>3.16</td>
</tr>
<tr>
<td></td>
<td>3.15</td>
</tr>
<tr>
<td></td>
<td>2.2</td>
</tr>
<tr>
<td></td>
<td>0.507</td>
</tr>
<tr>
<td></td>
<td>0.256</td>
</tr>
<tr>
<td></td>
<td>0.237</td>
</tr>
</tbody>
</table>

Teams of 1
Today’s lecture

• A look at PTX code
• Thread scheduling
Recapping from last time

• Nvcc tells us that our tiled kernel uses 30 registers
  ♦ 30K registers with a block size is 32 x 32
  ♦ These are single precision register counts
  ♦ We can run with 2 blocks /SM

• Hide arithmetic and memory latencies using fewer threads
  ♦ Unrolling increases ILP
  ♦ Unrolling increases register pressure, but reducing number of threads also lowers it
  ♦ ..by making better use of registers we can trade locality against parallelism
Hiding memory latency

- **Parallelism = latency \times throughput**

  Arithmetic: \(576 \text{ ops/SM} = 18\text{CP} \times 32/\text{SM/CP}\)
  Memory: \(150\text{KB} = \sim500\text{CP} \times 1100 \text{nsec} \times 150 \text{GB/sec}\)

- **How can we keep 150KB in flight?**
  - Multiple threads: \(\sim35,000\) threads @ 4B/thread
  - ILP (increase fetches per thread)
  - Larger fetches (64 or 128 bit/thread)
  - Higher occupancy

Copy 1 float /thread, need 100% occupancy

\[
\text{int } \text{indx} = \text{threadIdx.x} + \text{block } \times \text{blockDim.x}; \\
\text{float } a0 = \text{src}[\text{indx}]; \\
\text{dest}[\text{indx}] = a0;
\]

Copy 2 floats /thread, need 50% occ

\[
\text{float } a0 = \text{src}[\text{indx}]; \\
\text{float } a1 = \text{src}[\text{indx}+\text{blockDim.x}]; \\
\text{dest}[\text{indx}] = a0; \\
\text{dst}[\text{indx}+\text{blockDim.x}] = a1;
\]

Copy 4 floats /thread, need 25% occ

\[
\text{int } \text{indx} = \text{threadIdx.x} + 4 \times \text{block } \times \text{blockDim.x}; \\
\text{float } a[4]; \quad \text{// in registers} \\
\text{for}(i=0;i<4;i++) a[i]=\text{src}[\text{indx}+i*\text{blockDim.x}]; \\
\text{for}(i=0;i<4;i++) \text{dst}[\text{indx}+i*\text{blockDim.x}]=a[i];
\]

10/28/15 Scott B. Baden / CSE 260, UCSD / Fall '15
More about on chip memory

- 3 modes for shared memory/L1
  - No preference:
  - Favor shared memory:
  - Favor L1: x

- On GK210 (Sorken)
  - 96K+32K; 112K+16K; 80K+48K

- On GK110 (Stampede)
  - 32K+32K, 48+16K, 16+48K

- 48K read only data cache: program generated table of constants (lookup table)

- Shuffle instructions to move data without using shared memory
About PTX and PTXAS

- Nvcc translates cuda source into *PTX*, an intermediate form
- The PTXAS *back end compiler* Optimizes and assembles PTX into a binary object file
- PTX virtualizes registers, uses Static Single Assignment form (SSA) [en.wikipedia.org/wiki/Static_single_assignment_form](en.wikipedia.org/wiki/Static_single_assignment_form) (Prof. Jeanne Ferrante is a co-author)
- You’ll see many many registers in PTX code
- PTXAS maps virtual registers onto physical ones
- Nvcc --ptx reports # physical registers < # virtual registers
Looking at the PTX code

- See the example in $PUB/Examples/CUDA/incrArr
- Nvcc reports 6 registers, 4 registers for single precision
- Double precision values are contained in even-valued register pairs as are 64 bit addresses
- If we remove the conditional, 6 and 5 registers, respectively
- Single precision floating point constants need the ‘f’ qualifier as in
  \[ a[idx] = a[idx] + 1.0f; \]
- To read the ptx code, have the PTX ISA document handy

```
__global__ void incrementArrayOnDevice(_DOUBLE_ *a, int N) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx<N) a[idx] = a[idx] + 1;
}
```

docs.nvidia.com/cuda/parallel-thread-execution
The generated PTX code – function entry

- Global array argument is a 64 bit address, an unsigned integer
- The other values are standard 32 bit unsigned integers

```c
__global__ void incrementArrayOnDevice(_DOUBLE_ *a, int N) {
    ...
}
```

.visible – Externally visible symbol declaration
.entry - Kernel entry point and body, with optional parameters

‘f’ if single precision

```c
.visible .entry _Z22incrementArrayOnDevicePdii(
    .param .u64 _Z22incrementArrayOnDevicePfii_param_0,
    .param .u32 _Z22incrementArrayOnDevicePfii_param_1,
)
```
Virtualized registers

- In SSA form, every result is written to a new virtual register
- PTX manages arrays of registers using <> notation
- fd registers are twice as long as ‘f’ registers

```c
int idx = blockIdx.x*blockDim.x + threadIdx.x;
if (idx<N) a[idx] = a[idx]+ONE;
```

.reg - fast storage locations, 8, 16, 32, 64, 128 bits (predicates are 1 bit)
.reg .f64 %fd<3> declares 6 registers of DP floats %fd0 ... %fd5
.reg .s64 %rd<5> declares 5 registers of 64 bit signed integers
.reg .pred %<2>; .reg .pred %p<2>;
.reg .s32 %r<6>; .reg .s32 %r<6>;
.reg .f64 %fd<3>; .reg .f32 %f<3>;
.reg .s64 %rd<5>; .reg .s64 %rd<5>;

Double Single
The PTX code body

```c
__global__ void incrementArrayOnDevice(_DOUBLE_ *a, int N) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx<N) a[idx] = a[idx]+ONE;
}
```

```ptx
ld.param.u64    %rd1, [__Z22incrementArrayOnDevicePdii_param_0];
ld.param.u32    %r2, [__Z22incrementArrayOnDevicePdii_param_1];
mov.u32         %r3, %ctaid.x;    // Special read-only register, global
                   // block identifier, blockIdx.x
mov.u32         %r4, %ntid.x;    // blockDim.x
mov.u32         %r5, %tid.x;     // threadIdx.x
mad.lo.s32      %r1, %r4, %r3, %r5;   // compute IDX store in %r1
setp.ge.s32     %p1, %r1, %r2;   // Sets predicate register if r1>r2
@%p1 bra         BB6_2;        // Predicated execution, exits
cvta.to.global.u64 %rd2, %rd1;
mul.wide.s32     %rd3, %r1, 8;  // Computes the effective address
add.s64          %rd4, %rd2, %rd3;   // of a[idx]
ld.global.f64    %fd1, [%rd4];   // loads a[idx]
add.f64          %fd2, %fd1, 0d3FF0000000000000000;   // increments a[idx]
st.global.f64    [%rd4], %fd2;
```
How did I get 14 registers?

- Let’s look at the binary, to see the physical registers

```
cuobjdump -ptx -sass incr.o
```

```c
int idx = blockIdx.x*blockDim.x + threadIdx.x;
if (idx<N) a[idx] = a[idx]+1.0;
```

codefor sm_37 Function: _Z22incrementArrayOnDevicePdi

```assembly
/*0008*/    MOV R1, c[0x0][0x44]
/*0010*/    S2R R0, SR_CTAID.X  // Move special register to register
/*0018*/    S2R R3, SR_TID.X
/*0020*/    IMAD R0, R0, c[0x0][0x28], R3
/*0028*/    ISETP.GE.AND P0, PT, R0, c[0x0][0x148], PT
/*0030*/    @P0 BRA.U 0x70
/*0038*/    @!P0 MOV32I R3, 0x8
/*0048*/    @!P0 IMAD R4.CC, R0, R3, c[0x0][0x140]
/*0050*/    @!P0 IMAD.HI.X R5, R0, R3, c[0x0][0x144]
/*0058*/    @!P0 LD.E.64 R2, [R4]
/*0060*/    @!P0 DADD R2, R2, 1
/*0068*/    @!P0 ST.E.64 [R4], R2
/*0070*/    MOV RZ, RZ
/*0078*/    EXIT
```
Looking at the PTX code for Matrix Multiply

- See the example in $PUB/Examples/CUDA/mm-shmem-coalesce$
- Includes the ptx code
- Note typos on previous slide, which set up tx/ty, bx/by incorrectly

```c
__global__ mmpy(double *A, double *B, double *C){
    __shared__ double A[TW][TW], A[TW][TW];
    int tx = threadIdx.x, ty = threadIdx.y;
    int bx = blockIdx.x, by = blockIdx.y;
    int I = by*TW + ty, J = bx*TW+tx;
    double Cij = 0;
    for (int kk=0; kk<N/TW; kk++){
        As[ty][tx] = A[I*N + kk*TW+tx];
        Bs[ty][tx] = B[(kk*TW+ty)*N + J];
        __syncthreads();
        for (int k=0; k<TW; k++)
            Cij+= As[ty][k] * Bs[k][tx];
        __syncthreads();
        C[I*N + J] = Cij;
    }
}```
Today’s lecture

• A look at PTX code
• Thread scheduling
Thread scheduling

- Each SMX has 4 schedulers + 8 instruction dispatchers
- Each warp can support 2 independent instructions/cycle
- Each scheduler finds an eligible warp, 4 warps can be issued and scheduled simultaneously
- Multiple warps simultaneously active, hiding data transfer delays
Warp scheduler

- Assigns independent instructions for processing ... from same or different warp ... in same or different blocks
- Can dispatch 2 independent instructions/ cycle
- Can pair double precision instructions with others
- Scheduler makes inter-warp scheduling decisions: choose the best warp to go next if multiple candidates
- Hardware is free to assign blocks to any SMX, but once assigned to an SMX, block remains there
- Compiler uses static information about arithmetic instruction timings to inform the scheduler
- Requires some ILP for single precision, since 2 operations can be issued simultaneously
- All registers in all the warps are available, 0 overhead scheduling
- Overhead may be different when switching blocks
Mapping work onto processors

- A grid corresponds to a vectorizable loop
- From the software perspective a thread block …
  - is a single thread of vector instructions with a programmable vector length (the block size), allowing us to run on devices with different configurations
  - strip mines the loop
- Consider Vector- matrix multiply $A = BC$
  
  for $i = 0 : n - 1$
  
  for $j = 0 : n - 1$
  
Strip mining

• Partitioning the iteration space into chunks

\[
\text{for } i = 0 \text{ to } N-1 \\
a[i] = b[i] + c[i];
\]

\[
\text{for } j = 0 \text{ to } N-1 \text{ by } VL \\
\text{for } i = j \text{ to } \min(N, j+VL) - 1 \\
a[i] = b[i] + c[i];
\]

\[
\text{int } \text{idx} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}; \\
\text{if } (\text{idx}<N) \ a[\text{idx}] = a[\text{idx}]+1.f;
\]
Strip mining on the GPU

• Partitioning a thread block into warps corresponds to strip-mining into independent instruction streams

• Traditionally: independent instructions in the same instruction stream
  
  ```
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx]+1.f;
  
  for j = 0 to N-1 by VL
    for i = j to min(N, j+VL) – 1
      a[i] = b[i] + c[i];
  ```

Scott B. Baden / CSE 260, UCSD / Fall '15
Recapping: warp scheduling

- Multiple thread blocks may be assigned to an SM
- Each block divided into warps of 32 (SIMD) threads
  - A warp becomes eligible for execution when all its operands are available
  - Dynamic instruction reordering: eligible warps selected for execution using a prioritized scheduling policy
- Multiple warps simultaneously active, hiding data transfer delays
- Each warp is a wide SIMD instruction