Understanding GPU performance

How to get peak FLOPS (GPU version)

Kenjiro Taura
Contents

1 Recap: the CUDA programming model

2 An endeavor to nearly peak FLOPS (GPU version)
   - Latency
   - Looking into Low Level Code
   - Increasing the Throughput
   - Programming on GPU and CPU
   - A Deeper Look at Throughput
Contents

1 Recap: the CUDA programming model

2 An endeavor to nearly peak FLOPS (GPU version)
   • Latency
   • Looking into Low Level Code
   • Increasing the Throughput
   • Programming on GPU and CPU
   • A Deeper Look at Throughput
Recap: the CUDA programming model

- you write what a CUDA thread does in a device function (or a "CUDA kernel")

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

- launch a kernel with many threads

```
f<<<nb,bs>>>(...);
```
The peak FLOPS of GPUs

- **64 SP (32 DP) FMAs per cycle per SM** on latest architectures (Pascal and Volta)

(except for the much more special-purpose half-precision matrix-block multiply operations)

<table>
<thead>
<tr>
<th>arch</th>
<th>FMAs /cycle /SM (SP/DP)</th>
<th>freq GHz</th>
<th>SM</th>
<th>peak GFLOPS (SP/DP)</th>
<th>TDP W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal P100</td>
<td>64/32</td>
<td>1.328</td>
<td>56</td>
<td>9519/4760</td>
<td>300</td>
</tr>
<tr>
<td>Volta V100</td>
<td>64/32</td>
<td>1.530</td>
<td>80</td>
<td>15667/7833</td>
<td>300</td>
</tr>
</tbody>
</table>

- Pascal: $64 \times 1.328 \times 56 = 9519$
- Volta: $64 \times 1.530 \times 80 = 15667$
a CUDA thread

a kernel function describes a \textit{thread};
GPU execution hierarchy

32 threads (of consecutive threads IDs) make a *warp*;
32 threads (of consecutive threads IDs) make a *warp*; threads within a warp execute in a SIMD-like fashion; they execute the same instruction(s) in a cycle;
GPU execution hierarchy

several warps make a *thread block*;
several warps make a *thread block*; warps within a block are dispatched to the same *Streaming Multiprocessor (SM)*;
GPU execution hierarchy

warp in several thread blocks execute in an SM with a fine grain interleaving;
GPU execution hierarchy

and there are many SMs in a *GPU device*. 
GPU hierarchy vs. CPU hierarchy

- multiple devices (GPU or CPU) on a board
GPU hierarchy vs. CPU hierarchy

1. multiple devices (GPU or CPU) on a board
2. multiple cores (CPU core or SM) in a device
GPU hierarchy vs. CPU hierarchy

1. multiple devices (GPU or CPU) on a board
2. multiple cores (CPU core or SM) in a device
3. multiple hardware-scheduled threads within a core (HWT or thread blocks)
GPU hierarchy vs. CPU hierarchy

1. multiple devices (GPU or CPU) on a board
2. multiple cores (CPU core or SM) in a device
3. multiple hardware-scheduled threads within a core (HWT or thread blocks)
4. instruction level parallelism within a thread
GPU hierarchy vs. CPU hierarchy

- multiple devices (GPU or CPU) on a board
- multiple cores (CPU core or SM) in a device
- multiple hardware-scheduled threads within a core (HWT or thread blocks)
- instruction level parallelism within a thread
- parallelism within a single instruction (SIMD or warp)
What is the peak FLOPS (Pascal)?

- an SM has 2 processing units
- each processing unit can execute 1 warp in each cycle and dispatch $\leq 2$ instructions ($\leq 1$ instruction of the same type)
- each processing unit has, among others,
  - 32 SP (16 DP) FMA units (each capable of 1 FMA/cycle)
  - 8 LD/ST units (each capable of 8 bytes/cycle)
- $\Rightarrow$ peak FLOPS = 2 warps each executing 32 FMAs in a cycle

https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
What is the peak FLOPS (Volta)?

- an SM has 4 processing units
- each processing unit has, among others
  - 16 SP (8 DP) FMA units (each capable of 1 FMA/cycle)
  - 8 LD/ST units (each capable of 8 bytes/cycle)
- in each cycle, each processing unit can dispatch \( \leq 1 \) instructions from a warp
- \( \Rightarrow \) peak FLOPS = 4 warps each executing 16 FMAs in a cycle

https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
Recap: the CUDA programming model

An endeavor to nearly peak FLOPS (GPU version)
- Latency
- Looking into Low Level Code
- Increasing the Throughput
- Programming on GPU and CPU
- A Deeper Look at Throughput
Contents

1 Recap: the CUDA programming model

2 An endeavor to nearly peak FLOPS (GPU version)
   - Latency
   - Looking into Low Level Code
   - Increasing the Throughput
   - Programming on GPU and CPU
   - A Deeper Look at Throughput
An endeavor to nearly peak FLOPS

- the same “$x = ax + b$” iterations that we did on CPU

```c
__global__ void axpy_dev(long n, float a, float * X, float b) {
    for (long i = 0; i < n; i++) {
        X[j] = a * X[j] + b;
    }
}
```
A single CUDA thread (Pascal)

- run this with a single CUDA thread
  
  ```c
  axpy_dev<<1,1>>>(n, a, X, b);
  ```

- compile
  
  ```bash
  $ nvcc -o axpy.nvcc -O3 -Xptxas -O3 -x cu --generate-code
     arch=compute_60,code=sm_60 --compiler-options=-mavx2 axpy.cc
  ```

- run
  
  ```bash
  $ srun -p p -t 0:01:00 --gres=gpu:1 ./axpy.nvcc -a cuda
  algo = cuda
  bs = 1 (cuda block size)
  c = 1 (the number of variables to update in the inner loop)
  m = 1 (the number of FP numbers to update)
  n = 1000000 (the number of times to update each variable)
  FMAs = 1000000
  6690264 clocks, 2126592533 REF clocks, 1250939306 ns
  6.690264 clocks/iter, 2126.592533 REF clocks/iter, 1250.939306 ns/iter
  0.149471 FMAs/clock, 0.000470 FMAs/REF clock, 0.001599 GFLOPS
  ```
A single CUDA thread (Volta)

compile

```bash
$ make
nvcc -o axpy.nvcc -O3 -Xptxas -03 -x cu --generate-code
   arch=compute_70,code=sm_70 --compiler-options=-mavx2 axpy.cc
```

run

```bash
$ srun -p v -t 0:01:00 --gres=gpu:1 ./axpy.nvcc -a cuda
   algo = cuda
   bs = 1 (cuda block size)
   c = 1 (the number of variables to update in the inner loop)
   m = 1 (the number of FP numbers to update)
   n = 1000000 (the number of times to update each variable)
   FMAs = 1000000
   4438858 clocks, 4180065444 REF clocks, 1990502355 ns
   4.438858 clocks/iter, 4180.065444 REF clocks/iter, 1990.502355 ns/iter
   0.225283 FMAs/clock, 0.000239 FMAs/REF clock, 0.001005 GFLOPS
```
Latencies of common arithmetic

<table>
<thead>
<tr>
<th></th>
<th>Pascal</th>
<th>Volta</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer add</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>integer mul/mad</td>
<td>86</td>
<td>5</td>
</tr>
<tr>
<td>SP add/mul/fma</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>DP add/mul/fma</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

https://arxiv.org/abs/1804.06826

- measured SP FMA latencies
  - Pascal
    - 6.690264 clocks/iter, ...
  - Volta
    - 4.438858 clocks/iter, ...
Contents

1 Recap: the CUDA programming model

2 An endeavor to nearly peak FLOPS (GPU version)
   - Latency
   - Looking into Low Level Code
   - Increasing the Throughput
   - Programming on GPU and CPU
   - A Deeper Look at Throughput
Important options

- toolkit documentation: https://docs.nvidia.com/cuda/
Code generation options

- there are two kinds of machine code for GPUs
  - PTX (parallel thread execution) : a portable pseudo assembly
  - native code for each GPU model
- `--generate-code arch=compute_{nn},code=sm_{nn}` instructs nvcc to
  - generate PTX assuming a specific architecture
  - embed the native code for a specific GPU model
  - Pascal : \( nn=60 \)
  - Volta : \( nn=70 \)
- you can give `--generate-code` multiple times to generate an executable that runs on multiple architectures
- without any `--generate-code` (or similar) options, nvcc generates PTX for an old model (sm_30)
Looking into PTX and native code

<table>
<thead>
<tr>
<th></th>
<th>nvcc --ptx xxxx.cu</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxx.ptx</td>
<td>nvcc --fatbin xxxx.cu; cuobjdump --dump-sass xxxx.fatbin &gt; xxxx.asm</td>
</tr>
</tbody>
</table>

- **trade offs**
  - PTX is easier to deal with; you can embed comments
  - further optimizations apply in the PTX → native translation, so native code may be necessary for precise performance understanding
PTX for axpy

PTX (unrolled four times)

1  BB0_11:
2    fma.rn.f32 %f15,%f23,%f12,%f13;   // f15=f32*f12+f13
3    fma.rn.f32 %f16,%f15,%f12,%f13;
4    fma.rn.f32 %f17,%f16,%f12,%f13;
5    fma.rn.f32 %f23,%f17,%f12,%f13;
6    add.s64 %rd36,%rd36,4;
7    setp.lt.s64 %p7,%rd36,%rd15;
8    @%p7 bra BB0_11;
Native code for axpy

- native code for sm_60 (almost identical for sm_70)

```
1  /* 0328 */ FFMA R9, R9, R11.reuse, c[0x0][0x1a0]; /* 0x5180058006870909 */
2  /* 0330 */ FFMA R9, R9, R11.reuse, c[0x0][0x1a0]; /* 0x5180058006870909 */
3
4  ... 8 FFMA insns ...
5
6  /* 0390 */ IADD32I R6.CC, R6, 0x10; /* 0x1c10000001070606 */
7  /* 0398 */ FFMA R9, R9, R11, c[0x0][0x1a0]; /* 0x5180058006870909 */
8  /* 03a8 */ IADD.X R7, RZ, R7; /* 0x5c1008000077ff07 */
9  /* 03b0 */ FFMA R9, R9, R11.reuse, c[0x0][0x1a0]; /* 0x5180058006870909 */
10 /* 03b8 */ IADD RZ.CC, -R8, R6; /* 0x5c128000006708ff */
11 /* 03c8 */ FFMA R9, R9, R11.reuse, c[0x0][0x1a0]; /* 0x5180058006870909 */
12 /* 03d0 */ ISETP.LT.X.AND P1, PT, R7, R10, PT; /* 0x5b630b8000a7070f */
13 /* 03d8 */ FFMA R9, R9, R11, c[0x0][0x1a0]; /* 0x5180058006870909 */
14 /* 03e8 */ FFMA R9, R9, R11.reuse, c[0x0][0x1a0]; /* 0x5180058006870909 */
15 /* 03f0 */ { FFFMA R9, R9, R11, c[0x0][0x1a0]; /* 0x5180058006870909 */
16 /* 03f8 */   @P1 BRA 0x320; } /* 0xe2400ffff201000f */
```

- 16 FFMAs + 4 INTs + 1 branch
Making sense of the latency results

- the reported latency “6.690264 clocks/iter” on Pascal suggests the above unrolled loop runs in
  \[6.690264 \times 16 = 107.044224 = 6 \times 16 + 11\] cycles

- on Volta, it runs in
  \[4.438858 \times 16 = 71.021728 = 4 \times 16 + 7\] cycles

- both are at least close to integers
- I don’t know where these extra 11 / 7 cycles come from . . .
Debugging

- compile with no optimizations and with debug info
  
  $ nvcc -O0 -g -G -Xptxas -00 ...$

- use `cuda-gdb`; with slurm,

  $ srun -p p -t 0:30:00 --gres=gpu:1 --pty cuda-gdb executable

- consider also using `cuda-memcheck`
Contents

1 Recap: the CUDA programming model

2 An endeavor to nearly peak FLOPS (GPU version)
   - Latency
   - Looking into Low Level Code
   - Increasing the Throughput
   - Programming on GPU and CPU
   - A Deeper Look at Throughput
Increasing the **throughput**

- the gap to the peak (of a single SM) is enormous
  - Pascal: $= 6.690264 \times 64 = 428.17 \ldots$
  - Volta: $= 4.438858 \times 64 = 284.08 \ldots$

- the only way to go beyond this is, of course, to increase the parallelism!

- available tools
  1. increase ILP within a thread by increasing the number of variables to update (just as we did on CPU)
  2. increase the number of threads!
Increase ILP

- each thread updates $c$ variables

```c
float X[c];

/* initialize X */
for (long i = 0; i < n; i++)
    for (long j = 0; j < c; j++)
        X[j] = a * X[j] + b;
```

- still launch just a single thread

```c
axpy_dev<<<1,1>>>(n, a, X, b);
```
Increase the number of threads (in a single SM)

- this is the GPU way of increasing parallelism

```c
axpy_dev<<<1, bs>>>(n, a, X, b);
```

- the zigzag pattern reflects the number of processing units (Pascal: 2, Volta: 4)
Contents

1 Recap: the CUDA programming model

2 An endeavor to nearly peak FLOPS (GPU version)
   - Latency
   - Looking into Low Level Code
   - Increasing the Throughput
   - Programming on GPU and CPU
   - A Deeper Look at Throughput
CUDA unifies SIMD and thread programming

- commonality: both CPU and GPU employ a SIMD-like execution machinery, for a good peak performance with a modest decode/dispatch throughput

```
f<<<1,32>>>(x);
```

a warp (32 threads)
CUDA unifies SIMD and thread programming

- **Commonality:** Both CPU and GPU employ a SIMD-like execution machinery, for a good peak performance with a modest decode/dispatch throughput.

- **Difference:** In CUDA, exploiting SIMD (warp) is almost automatic; just increase the number of threads per block and GPU executes 32 threads as a warp (SIMT).

```c
f<<<1,32>>>(x);
```
CUDA relieves you from SIMD programming

- GPU handles many things that hinder vectorization on CPU

<table>
<thead>
<tr>
<th>issues</th>
<th>how you do it on CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>branches</td>
<td>predicated execution</td>
</tr>
<tr>
<td>loops whose trip counts vary across threads</td>
<td>predicated execution</td>
</tr>
<tr>
<td>non-contiguous data accesses</td>
<td>gather/scatter</td>
</tr>
<tr>
<td>atomic update</td>
<td>N/A</td>
</tr>
</tbody>
</table>

- a great boon to the programmer
- note that the performance penalty *is* still visible on GPUs too
GPU core is *highly* multithreaded

- commonality: a single thread (warp) performance is often limited by *latencies*
GPU core is *highly* multithreaded

- commonality: a single thread (warp) performance is often limited by *latencies*

- commonality: both CPU cores and GPU SMs are *multithreaded (SMT)*; i.e., each core (SM) can simultaneously execute multiple threads (warps), switching between them every few cycles

| recent Xeon’s | 2 |
| Xeon Phi | 4 |
| Power 9 | 8 |
| Nvidia GPUs | 64 |
GPU core is *highly* multithreaded

- commonality: a single thread (warp) performance is often limited by *latencies*
- commonality: both CPU cores and GPU SMs are *multithreaded (SMT)*; i.e., each core (SM) can simultaneously execute multiple threads (warps), switching between them every few cycles

- difference: GPU SMs are *highly* multithreaded
  - each SM supports up to 2048 threads (64 warps)

| recent Xeon’s | 2 |
| Xeon Phi      | 4 |
| Power 9       | 8 |
| Nvidia GPUs   | 64 |
1 Recap: the CUDA programming model

2 An endeavor to nearly peak FLOPS (GPU version)
   - Latency
   - Looking into Low Level Code
   - Increasing the Throughput
   - Programming on GPU and CPU
   - A Deeper Look at Throughput
Making sense of the gap to the peak (Pascal)

- from the latency (6.690264 cycles/FMA), we hope that, even with $c = 1$ (each thread updating just a single variable), they reach 64 FMAs/cycle with

$$64 \times 6.690264 = 428.17 \ldots \rightarrow 448 \text{ threads (14 warps)}$$

- the actual peak throughput plateaus at $\approx 47$ FMAs/sec
Making sense of the gap to the peak (Pascal)

- Pascal dispatches both SP arithmetic and INT arithmetic to the same execution unit (CUDA Core)
- overhead of integer instructions are often not negligible on Pascal
- the number of integer instructions is a good predictor of the throughput limit

<table>
<thead>
<tr>
<th>c</th>
<th>SP insns</th>
<th>INT insns</th>
<th>FMAs/cycle (predicted)</th>
<th>FMAs/cycle (measured)</th>
<th>the thread count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>4</td>
<td>$64 \times 16/20 = 51.2$</td>
<td>47.19</td>
<td>448</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>5</td>
<td>$64 \times 32/37 = 55.35$</td>
<td>55.21</td>
<td>512</td>
</tr>
<tr>
<td>3</td>
<td>48</td>
<td>5</td>
<td>$64 \times 48/53 = 57.96$</td>
<td>57.91</td>
<td>384</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>5</td>
<td>$64 \times 64/69 = 59.36$</td>
<td>59.33</td>
<td>384</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>5</td>
<td>$64 \times 20/25 = 51.2$</td>
<td>50.72</td>
<td>448</td>
</tr>
<tr>
<td>6</td>
<td>24</td>
<td>5</td>
<td>$64 \times 24/29 = 52.96$</td>
<td>52.93</td>
<td>384</td>
</tr>
<tr>
<td>7</td>
<td>28</td>
<td>5</td>
<td>$64 \times 28/33 = 54.30$</td>
<td>54.23</td>
<td>512</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>5</td>
<td>$64 \times 32/37 = 55.35$</td>
<td>55.29</td>
<td>384</td>
</tr>
</tbody>
</table>
Making sense of the zigzag pattern

- **Pascal (2 processing units)**

<table>
<thead>
<tr>
<th>threads</th>
<th>warps</th>
<th>cycles</th>
<th>FMAs/cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>384</td>
<td>12</td>
<td>8413085</td>
<td>45.64</td>
</tr>
<tr>
<td>416</td>
<td>13</td>
<td>9502041</td>
<td>43.78</td>
</tr>
<tr>
<td>448</td>
<td>14</td>
<td>9491826</td>
<td>47.19</td>
</tr>
<tr>
<td>480</td>
<td>15</td>
<td>10863789</td>
<td>44.18</td>
</tr>
<tr>
<td>512</td>
<td>16</td>
<td>10862553</td>
<td>47.13</td>
</tr>
<tr>
<td>544</td>
<td>17</td>
<td>12944833</td>
<td>42.02</td>
</tr>
<tr>
<td>576</td>
<td>18</td>
<td>12910888</td>
<td>44.61</td>
</tr>
</tbody>
</table>

- **Volta (4 processing units)**

<table>
<thead>
<tr>
<th>threads</th>
<th>warps</th>
<th>cycles</th>
<th>FMAs/cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>384</td>
<td>12</td>
<td>6648872</td>
<td>57.75</td>
</tr>
<tr>
<td>416</td>
<td>13</td>
<td>8666550</td>
<td>48.00</td>
</tr>
<tr>
<td>448</td>
<td>14</td>
<td>8693014</td>
<td>51.53</td>
</tr>
<tr>
<td>480</td>
<td>15</td>
<td>8704994</td>
<td>55.14</td>
</tr>
<tr>
<td>512</td>
<td>16</td>
<td>8696415</td>
<td>58.87</td>
</tr>
<tr>
<td>544</td>
<td>17</td>
<td>10878118</td>
<td>50.00</td>
</tr>
<tr>
<td>576</td>
<td>18</td>
<td>10877196</td>
<td>52.95</td>
</tr>
<tr>
<td>608</td>
<td>19</td>
<td>10879399</td>
<td>55.88</td>
</tr>
<tr>
<td>640</td>
<td>20</td>
<td>10885095</td>
<td>58.79</td>
</tr>
</tbody>
</table>

if the number of warps on SM is not a multiple of the number of processing units, a slight load imbalance will result among them.

on Pascal, the clocks with $(2k - 1)$ and $2k$ warps are almost exactly the same.

on Volta, the clocks with $(4k - 3) \cdots 4k$ warps are almost exactly the same.
the motto:

- a single thread is slow (has a long latency → low throughput)
- “have more threads, until you hit the peak (FLOPS)”

but then, can we perform arbitrary computation with very long latency at close to peak FLOPS, just by throwing more threads?
Limit of parallelism per SM

- of course not
  - you may hit the throughput limit of other units first (esp. LD/ST units)
  - an SM has a limit on the number of simultaneously live threads
two major resource constraints

- register capacity
- shared memory (covered later)

<table>
<thead>
<tr>
<th>registers per thread</th>
<th>Pascal</th>
<th>Volta</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared memory</td>
<td>64K × 4 bytes</td>
<td>64K × 4 bytes</td>
</tr>
<tr>
<td></td>
<td>64KB</td>
<td>96KB</td>
</tr>
</tbody>
</table>

registers: a thread block that has any live (= started but not finished) thread occupies this many registers:

\[
\text{registers per thread} \times \text{the number of threads per block}
\]

⇒ the number of simultaneously live thread blocks

\[
\leq \left\lfloor \frac{\text{register capacity}}{\text{registers per thread} \times \text{the number of threads per block}} \right\rfloor
\]