TENSILE
HIGH-PERFORMANCE GEMM ON GPUS
IS LIKE A 3-YEAR OLD
David E Tanner
GEMM API encapsulates trillions of problems, all of which behave differently on GPUs.
  – Precisions, Transpose A, Transpose B, M, N, K, Strides

Many problem types behave similar to GEMM.
  – $C_{ij} = \sum_k A_{ik} \times B_{kj}$ (gemm NN)
  – $C_{ijk} = \sum_l A_{ilk} \times B_{jlk}$ (batched gemm NT)
  – $C_{ijk} = \sum_{lmn} A_{ilmnk} \times B_{jlmnk}$ (batched gemm w/ 3D summation)
  – $C_{ijk} = \sum_{lmn} A_{inlkm} \times B_{mijnkn}$ (batched gemm w/ 3D summation different data layout)

Goal: auto-generate kernels with peak performance
  – For all problem types.
  – For all problem sizes.
  – On all GPUs.
  – Multiple languages: OpenCL, HIP, Assembly.
GPU PERFORMANCE PARAMETERS
VEGA10 FRONTIER EDITION

- **Compute Throughput**
  - 13.1 TFlops = 2 (flops/cycle)*64(CUs)*64(lanes/CU)*1600MHz

- **LDS Bandwidth**
  - TB/s

- **Caches**
  - L2 shared by all CUs
  - L1 dedicated to CU

- **Global Memory Bandwidth**
  - 480 GB/s
  - coalescing

- **Global Memory Latency**
  - hundreds of cycles
  - hide using CU-occupancy or ILP

- **CU Occupancy**
  - limited resources (VGPRs, LDS) per CU

- **Whole-GPU Occupancy**
  - hundreds of work-groups

- **LDS Latency**
  - tens of cycles
  - hide using CU-occupancy or ILP

- **Instruction Divergence**
  - all threads within workgroup do same instruction else need to compute and apply execution masks

- **Instruction Throughput**
  - gemm requires 2*M*N*K instructions, all extras hurt efficiency
  - minimize instructions which don’t count
  - maximize dual-issuing of instructions which don’t count with instructions that do; must be from different wavefronts
    - VALU, SALU, LDS, global memory, branch

- **Power**
  - VALU, LDS, memory, caches
Tiling at all memory levels to read from lower-bandwidth memory less and from higher-bandwidth memory more to prevent bandwidth from being bottleneck.

1. **Grid of 16x8 Workgroups**
   - Read from global once, store in L2
   - Read from L2 and store into LDS 8-16x

2. **Workgroup of 16x16 Threads**
   - Read from LDS and store in VGPRs 16x

3. **Thread has 8x8 elements**
   - 8 A VGPRs
   - 8 B VGPRs
   - 64 C VGPRs, 64 FMAs
THREA-D-TILE SUB-ITERATION
WHERE WE WANT TO GET TO

// read 8 A elements
ds_read_b128 // 16 threads reading exact same address of LDS; coalesced.
ds_read_b128
ds_read_b128

// read 8 B elements
ds_read_b128
ds_read_b128
ds_read_b128

s_waitcnt // wait for loads

// 64 MACs
v_mac_f32
v_mac_f32
v_mac_f32
v_mac_f32
+ 60 more

(2) Workgroup of 16x16 Threads
• Read from LDS and store in VGPRs 16x

(3) Thread has 8x8 elements
• 8 A VGPRs
• 8 B VGPRs
• 64 C VGPRs, 64 FMAs
SUMMATION LOOP

no prefetching

BEGIN LOOP
read global
wait global
barrier
write lds
barrier
read lds 0
wait lds 0
MACs 0
read lds 1
wait lds 1
MACs 1
...
read lds N-2
wait lds N-2
MACs N-2
read lds N-1
wait lds N-1
MACs N-1
END LOOP

write vgprs to C

BEGIN LOOP
read global iter 1
wait global read
write lds
barrier
read lds 0
wait lds read 0
MACs 0
read lds 2
wait lds read 1
MACs 1
...
read lds N-1
wait global read
write lds
swap lds red / black ptrs
wait read lds N-2
MACs N-2
wait write lds & N-1
barrier
read lds iter 1 subiter 0
MACs N-1
END LOOP

write vgprs to C

prefetch global
prefetch local
GLOBAL TO LDS

Don’t Transpose Data

K=256

M=512

Don’t Transpose Data

M=512

K=256

// read 4 A elements
// 512 bytes coalesced x8
flat_load_dwordx4

// read 4 B elements
flat_load_dwordx4

// write A to LDS
ds_write_b128

// write B to LDS
ds_write_b128

Do Transpose Data

// read 4 A elements
// 32 bytes coalesced x128
flat_load_dwordx4

// write A to LDS
ds_write_b32
ds_write_b32
ds_write_b32
ds_write_b32

// write B to LDS
ds_write_b32
ds_write_b32
ds_write_b32
ds_write_b32
EDGES WITHOUT DIVERGENCE

- Protect against reading out of bounds M, N
- Protect against reading out of bounds K
  Tail loop to handle K % UNROLL

coalesced

redundant reads
KERNEL PARAMETERS

1) WorkGroup, LocalSplitU
2) ThreadTile
3) VectorWidth
4) GlobalSplitU
5) GlobalSplitUWGM
6) PrefetchGlobalRead
7) PrefetchLocalRead
8) WorkGroupMapping
9) LoopUnroll
10) NumLoadsCoalesced
11) GlobalReadCoalesceGroup
12) GlobalReadCoalesceVector
13) KernelLanguage
14) NonTemporal

BEGIN LOOP
load global iter 1
wait lds 1
MACs 0
read lds 2
wait lds read 1
MACs 1
...
END LOOP
write vgprs to C

calculate addresses
prefetch iter 0
wait global read
write lds
swap lds red / black ptrs
wait read lds N-2
MACs N-2
barrier
read lds iter 1 subiter 0
MACs N-1
write lds N-1
wait global read
write lds
Kernel Parameters

- Global SplitU
- Global SplitU Map
- Workgroup Map
- Local SplitU
- Vector Width
- Workgroup
- Prefetch LDS
- Prefetch Global
- Num Loads Coalesced
- Read Vectors
- Thread Tile
- Loop Unroll

GPU Performance

- GPU Occupancy
- Caching
- LDS Bandwidth
- Global Mem Bandwidth
- Instruction Throughput
- LDS Latency
- Global Mem Latency

<table>
<thead>
<tr>
<th>Num Threads</th>
<th>LDS</th>
<th>Registers</th>
<th>Occupancy</th>
</tr>
</thead>
</table>

KERNEL PARAMETERS AFFECT GPU PERFORMANCE
**PERFORMANCE EXPERIMENTS**

Set of 100 kernels
- ThreadTile = 8x8, 8x4, 8x2, 4x8, 4x4, 4x2, 2x8, 2x4, 2x2
- WorkGroup = 16x16x1, 16x8x2, 8x16x2, 8x8x4
- GlobalSplitU = 1, 2, 4, 6, 8
- DepthU = 8
- VectorWidth = max
- PrefetchGlobalRead = True
- PrefetchLocalRead = True
- WorkGroupMapping = 8

Benchmark kernels against 2D range of sizes for sgemm NT
- M = 16 – 5632; N = 16 – 5632; K = 3104  (moderate)
- M = 128 – 100,000; N = 16 – 256; K = 3104  (skinny N)
- M = 64 – 512; N = 64 – 512; K = 1,048,576  (small MxN) GSU=16, 32, 64, 128

Analyze performance and kernel properties for each data point.
<table>
<thead>
<tr>
<th>N</th>
<th>TFlops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,131</td>
<td>9,158</td>
</tr>
<tr>
<td>1,134</td>
<td>9,552</td>
</tr>
<tr>
<td>1,156</td>
<td>9,703</td>
</tr>
<tr>
<td>1,162</td>
<td>9,824</td>
</tr>
<tr>
<td>1,164</td>
<td>9,856</td>
</tr>
<tr>
<td>1,166</td>
<td>9,914</td>
</tr>
<tr>
<td>1,168</td>
<td>9,966</td>
</tr>
<tr>
<td>1,172</td>
<td>10,133</td>
</tr>
<tr>
<td>1,174</td>
<td>10,161</td>
</tr>
<tr>
<td>1,176</td>
<td>10,268</td>
</tr>
<tr>
<td>1,179</td>
<td>10,310</td>
</tr>
<tr>
<td>1,182</td>
<td>10,358</td>
</tr>
<tr>
<td>1,184</td>
<td>10,411</td>
</tr>
<tr>
<td>1,186</td>
<td>10,467</td>
</tr>
<tr>
<td>1,188</td>
<td>10,521</td>
</tr>
<tr>
<td>1,190</td>
<td>10,577</td>
</tr>
<tr>
<td>1,192</td>
<td>10,608</td>
</tr>
<tr>
<td>1,194</td>
<td>10,651</td>
</tr>
<tr>
<td>1,196</td>
<td>10,694</td>
</tr>
<tr>
<td>1,198</td>
<td>10,732</td>
</tr>
<tr>
<td>1,200</td>
<td>10,775</td>
</tr>
<tr>
<td>1,202</td>
<td>10,824</td>
</tr>
<tr>
<td>1,204</td>
<td>10,871</td>
</tr>
<tr>
<td>1,206</td>
<td>10,914</td>
</tr>
<tr>
<td>1,208</td>
<td>10,961</td>
</tr>
<tr>
<td>1,210</td>
<td>11,005</td>
</tr>
<tr>
<td>1,212</td>
<td>11,049</td>
</tr>
<tr>
<td>1,214</td>
<td>11,093</td>
</tr>
<tr>
<td>1,216</td>
<td>11,137</td>
</tr>
<tr>
<td>1,218</td>
<td>11,183</td>
</tr>
<tr>
<td>1,220</td>
<td>11,227</td>
</tr>
<tr>
<td>1,222</td>
<td>11,272</td>
</tr>
<tr>
<td>1,224</td>
<td>11,316</td>
</tr>
<tr>
<td>1,226</td>
<td>11,360</td>
</tr>
<tr>
<td>1,228</td>
<td>11,405</td>
</tr>
</tbody>
</table>

**Peak:** 13 TFlops

**K = 3104**
Conclusion 2: Performance depends on size, not shape (unless skinny).
LARGE & SKINNY

GFlops

N

16
32
64
128
192
256

90% of highest

2X

M

K = 3104
SKINNY DIMENSIONS HURT PERFORMANCE

Matrix A
- Loaded once
- Used in 32 MACs

Matrix B
- Loaded once
- Used in 8 MACs

8x32

32x32
## Conclusion

For peak, MxN has to be at least 128x256, and M*N*K has to be sufficiently large (~3000^3).
EXPERIMENT 2 - DEEPBENCH

- Set of thousands of kernels.
- Benchmark kernels against ~250 problem sizes of DeepBench.
- Analyze speedup of kernel tuned for exact problem size vs “fastest” 128x128 tile kernel.

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>8457</td>
<td>1760</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>500000</td>
</tr>
<tr>
<td>512</td>
<td>8</td>
<td>500000</td>
</tr>
<tr>
<td>1024</td>
<td>1</td>
<td>512</td>
</tr>
<tr>
<td>1024</td>
<td>32</td>
<td>512</td>
</tr>
<tr>
<td>1760</td>
<td>128</td>
<td>1760</td>
</tr>
<tr>
<td>8448</td>
<td>48000</td>
<td>2816</td>
</tr>
</tbody>
</table>
SPEEDUP OF DEEPBENCH SIZES

100% means equal performance
MANY PARAMETERS

- ThreadTile: 2x2, 4x2, 4x4, 6x3, 3x3, 8x4, 4x8, 6x8, 8x2, 8x6, 8x8
- WorkGroup: 8x8x2, 8x8x4, 16x4x4, 8x4x8, 8x4x8, 8x8x2, 16x8x2, 12x16x1, 8x12x2, 16x16x1, 32x2x4, 16x2x8, 32x4x2
- GlobalSplitU: 1, 2, 4, 8, 16, 32
- DepthU: 8, 16, 24, 32, 64

- 68 winning kernels
- 4,290 permutations amongst winning parameters (even more explored)
<table>
<thead>
<tr>
<th>TENSILE</th>
<th>PUBLIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.8</td>
<td>16.7</td>
</tr>
<tr>
<td>16.4</td>
<td>15.5</td>
</tr>
<tr>
<td>15.0</td>
<td>14.5</td>
</tr>
<tr>
<td>14.0</td>
<td>13.6</td>
</tr>
<tr>
<td>13.1</td>
<td>12.8</td>
</tr>
<tr>
<td>12.5</td>
<td>12.1</td>
</tr>
<tr>
<td>11.6</td>
<td>11.3</td>
</tr>
<tr>
<td>10.9</td>
<td>10.6</td>
</tr>
<tr>
<td>10.2</td>
<td>9.9</td>
</tr>
<tr>
<td>9.5</td>
<td>9.2</td>
</tr>
<tr>
<td>8.8</td>
<td>8.4</td>
</tr>
<tr>
<td>7.9</td>
<td>7.5</td>
</tr>
<tr>
<td>7.0</td>
<td>6.6</td>
</tr>
<tr>
<td>6.1</td>
<td>5.8</td>
</tr>
<tr>
<td>5.2</td>
<td>4.9</td>
</tr>
<tr>
<td>4.1</td>
<td>3.8</td>
</tr>
<tr>
<td>3.4</td>
<td>3.1</td>
</tr>
<tr>
<td>2.6</td>
<td>2.3</td>
</tr>
<tr>
<td>1.8</td>
<td>1.5</td>
</tr>
<tr>
<td>1.1</td>
<td>0.8</td>
</tr>
<tr>
<td>0.4</td>
<td>0.1</td>
</tr>
</tbody>
</table>

**Notes:**
- The values in the table represent tensile and public data for various measurements.
- The measurements appear to be related to material properties or similar metrics.
HOW DO WE MAKE SMALL SKINNY FASTER?
IF WE CAN'T USE LARGE TILE, PRECEDING MEMORY MUST BE FASTER

Grid WorkGroup ThreadTile
Global L2 LDS Registers

Registers

A B C

A B C
Thank you
SUPPORTED PROBLEM TYPES

Example:

\[ C_{ijk} = \alpha \sum_{lmn} A_{inlkm} \ast B_{mjln} + \beta C_{ijk} \]

Tensor indices are ordered shortest to largest stride, i.e., the zeroth index/dimension has a stride of 1 (typically).

- **C indices** are labeled alphabetically starting with “i”.
- **Summation indices** are labeled alphabetically picking up where C indices left off.
- A, B indices are then labeled according to which summation or C index they correspond to.

- Tensile kernel will employ:
  - Enough work-groups to cover all the dimensions of C
  - Enough nested loops to carry out the multi-dimensional summation.

- A kernel for a given problem type will give correct answer for any problem sizes.
LARGE & SKINNY - 90% OF HIGHEST
MINIMUM PROBLEM SIZE FOR HIGH PERFORMANCE

<table>
<thead>
<tr>
<th>N</th>
<th>TFlops</th>
<th>90%</th>
<th>M</th>
<th>MT</th>
<th>Flops/Byte</th>
<th># WG</th>
<th>WG/CU</th>
<th>CU Occ</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2.3</td>
<td>2.0</td>
<td>7168</td>
<td>64x32</td>
<td>10.6</td>
<td>112</td>
<td>1.75</td>
<td>4</td>
</tr>
<tr>
<td>32</td>
<td>4.3</td>
<td>3.8</td>
<td>5888</td>
<td>32x32</td>
<td>8</td>
<td>184</td>
<td>1.3</td>
<td>8</td>
</tr>
<tr>
<td>64</td>
<td>8.7</td>
<td>8.0</td>
<td>11776</td>
<td>128x32</td>
<td>12.8</td>
<td>184</td>
<td>1.3</td>
<td>4</td>
</tr>
<tr>
<td>128</td>
<td>11.5</td>
<td>10.3</td>
<td>15488</td>
<td>64x128</td>
<td>21.3</td>
<td>242</td>
<td>3.8</td>
<td>3</td>
</tr>
<tr>
<td>192</td>
<td>11.0</td>
<td>10.5</td>
<td>13568</td>
<td>64x64</td>
<td>16</td>
<td>636</td>
<td>5.6</td>
<td>4</td>
</tr>
<tr>
<td>256</td>
<td>11.8</td>
<td>10.6</td>
<td>10112</td>
<td>128x64</td>
<td>21.3</td>
<td>316</td>
<td>4.9</td>
<td>3</td>
</tr>
</tbody>
</table>

Conclusion 3: To achieve peak performance, need sufficient work to fill GPU with large tiles.

Conclusion 4: Performance of small or skinny sizes bottlenecked by L2 bandwidth.

Vega10 @ 27.3 Flops / byte = (13200 Gflop/sec) / (483 Gbyte/sec) L2 is 2X faster

“skinny”

Not “skinny”

Large Tile

Overfill CUs