Name _____

# GPU Microarchitecture
# EE 7722
# Take-Home Final Examination

Thursday, 1 May 2019 to Sunday, 5 May 2019

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

| | | |
|---|---|---|
| Problem 1 | _____ | (50 pts) |
| Problem 2 | _____ | (50 pts) |
| Exam Total | _____ | (100 pts) |

Alias _____

*Good Luck!*

Problem 1: [50 pts] In Homework 3 there were several variations on a kernel that performed sparse work. The goal was to find efficient ways to concentrate work so that all threads in a warp would keep busy. In the original assignment package the baseline kernel, `mxv`, would not try to keep all threads in a warp busy. That kernel did poorly as sparsity dropped. However, the kernel had one major flaw which has since been fixed: data was read and written even if no operation was to be performed. (The `mxv` kernel was fixed in commit 2b32ea6.)

If execution time were determined only by data volume all kernels in the Homework 3 assignment code would have the same execution time since they each read and write the same amount of data. Two other factors can determine execution time: instruction throughput and interval latency.

(*a*) Model the code in `mxv` using convenient simplifying assumptions. To the same for the compression kernels: `mvx_pfx_ballot`, which at each iteration concentrates work found in all threads in a warp, and for `mvx_pfx_prob1`, in which each warp concentrates work over multiple iterations until there 32 work items. Avoid tedious models. Model intervals with and without work and estimate run time using these.

The model should show that at a high density (labeled `work` in the tables) the code should be bandwidth-limited and so `mxv` should perform as well as the compression kernels. Find a value of density (use $d$) at which each compression kernel is faster than `mxv` (based on the model). Explain why they are faster.

Using the model show that at lower densities latency should dominate and so the compression kernels should dominate.

(*b*) On the next page data from runs of `mxv` and two compression kernels are shown: `mxv_pfx_ballot` and a tuned Homework 3 solution. These runs show that `mxv` does well at higher densities.

The first set of runs were configured to maximize the number of resident warps for each kernel. The second set were configured for active 32 warps per MP, and the third set for 8 warps per MP. For each set look at the speedup of the `prob1` kernel over `mxv` at density .021. In the first set the speedup is a mere $\frac{1247\,\mu s}{1187\,\mu s} = 1.05$. The second and third are $\frac{1273\,\mu s}{1167\,\mu s} = 1.09$ and $\frac{2746\,\mu s}{1762\,\mu s} = 1.56$.

At density 1 the speedups are 1.00, 1.00, and 1.03, indicating that the `prob1` kernel has at best a small advantage at high densities and fewer warps.

Use the model to explain the speedups and other data. In particular explain why the speedup at low density is so much higher for the 8-warp configuration than the configurations with more warps.

The data below was run on `nereid.ece.lsu.edu` using commit 2e397d8.

```
GPU 1: GeForce GTX 1080 @ 1.73 GHz WITH 8119 MiB GLOBAL MEM
GPU 1: L2: 2048 kiB   MEM<->L2: 320.3 GB/s
GPU 1: CC: 6.1  MP: 20  CC/MP: 128  DP/MP:  4  TH/BL: 1024
GPU 1: SHARED: 49152 B/BL  98304 B/MP  CONST: 65536 B  # REGS: 65536
GPU 1: PEAK: 4438 SP GFLOPS  139 DP GFLOPS  COMP/COMM:  55.4 SP  3.5 DP
Using GPU 1

CUDA Kernel Resource Usage:
For mxv:
   32800 shared, 336 const, 0 loc, 27 regs; 1024 max threads per block.
For mxv_pfx_ballot:
   34852 shared, 336 const, 0 loc, 40 regs; 1024 max threads per block.
For mxv_pfx_prob1:
   40992 shared, 336 const, 0 loc, 40 regs; 1024 max threads per block.
Matrix size: 8 x 8.  Vectors: 20971520.
Kernel mxv:
                              R-Eff-% -L2-Cache-- ---DRAM----
wp ac  work  t/μs I/op SM eff  Ld  St  Rd θ  Wr θ  Rd θ  Wr θ GB/s Data BW Util-
32 64 1.000  7713  2.2 100.0% 100  50  97.9 174.0  97.9  87.0  185 *******
32 64 0.750  6611  2.9  81.4% 100  54  88.8 140.4  88.8  76.2  165 ******
32 64 0.500  4940  4.1  64.5% 100  67  84.9 101.9  84.9  68.0  153 ******
32 64 0.250  4326  7.1  52.0% 100  86  58.2  44.9  58.2  38.9   97 ***
32 64 0.021  1247 11.8  48.4% 100 100  78.8  11.6  78.8  12.2   90 ***
32 64 0.004   659  5.1  48.6% 100 100 131.7   4.3 131.6   5.7  136 *****
32 64 0.000   479  2.1  53.3% 100  -5 175.2   0.0 175.2   2.1  175 *******
Kernel mxv_pfx_ballot:
                              R-Eff-% -L2-Cache-- ---DRAM----
wp ac  work  t/μs I/op SM eff  Ld  St  Rd θ  Wr θ  Rd θ  Wr θ GB/s Data BW Util-
24 48 1.000  7532  2.9  74.1% 100  48 100.2 183.8 100.2  89.1  189 *******
24 48 0.750  6751  3.1  70.8% 100  49  87.4 151.9  87.1  74.6  162 ******
24 48 0.500  4877  3.3  66.4% 100  49  86.6 141.4  86.3  68.9  155 ******
24 48 0.250  3277  3.8  55.9% 100  47  77.7 108.1  77.5  51.4  128 *****
24 48 0.021  1134  6.6  13.3% 100  34  89.4  37.8  89.3  13.4   99 ****
24 48 0.004   951  7.5   3.9% 100  13  95.4  23.3  95.3   4.0   94 ***
24 48 0.000   599  6.9   1.1% 100  39 140.1   0.0 140.1   1.7  140 *****
Kernel mxv_pfx_prob1:
                              R-Eff-% -L2-Cache-- ---DRAM----
wp ac  work  t/μs I/op SM eff  Ld  St  Rd θ  Wr θ  Rd θ  Wr θ GB/s Data BW Util-
24 48 1.000  7744  2.4  92.3% 100  50  97.5 173.3  97.5  86.7  184 *******
24 48 0.750  7695  2.5  91.5% 100  50  76.3 130.8  76.3  65.4  142 *****
24 48 0.500  6755  2.6  89.9% 100  50  62.1  99.4  62.1  49.8  112 ****
24 48 0.250  5716  2.8  85.5% 100  50  44.1  58.7  44.0  29.5   73 **
24 48 0.021  1187  3.3  57.9% 100  50  82.9  24.3  82.8  12.8   95 ***
24 48 0.004   616  2.9  50.7% 100  51 140.9   9.3 140.7   5.9  145 *****
24 48 0.000   516  2.7  77.8% 100  64 162.6   0.0 162.6   2.0  163 ******
```

Kernel mxv:

| | | | | | R-Eff-% | | -L2-Cache-- | | ---DRAM---- | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| wp | ac | work | t/$\mu$s | I/op | SM eff | Ld | St | Rd $\theta$ | Wr $\theta$ | Rd $\theta$ | Wr $\theta$ | GB/s | Data BW Util- |
| 32 | 32 | 1.000 | 7691 | 2.2 | 100.0% | 100 | 50 | 98.2 | 174.5 | 98.2 | 87.3 | 185 | ******* |
| 32 | 32 | 0.750 | 6448 | 2.9 | 81.4% | 100 | 54 | 91.1 | 143.9 | 91.1 | 78.1 | 169 | ****** |
| 32 | 32 | 0.500 | 4809 | 4.1 | 64.5% | 100 | 67 | 87.2 | 104.7 | 87.2 | 69.9 | 157 | ****** |
| 32 | 32 | 0.250 | 3564 | 7.1 | 52.0% | 100 | 86 | 70.6 | 54.5 | 70.6 | 47.3 | 118 | **** |
| 32 | 32 | 0.021 | 1273 | 11.8 | 48.4% | 100 | 100 | 77.2 | 11.3 | 77.2 | 12.0 | 89 | *** |
| 32 | 32 | 0.004 | 751 | 5.1 | 48.6% | 100 | 100 | 115.4 | 3.7 | 115.4 | 5.0 | 119 | **** |
| 32 | 32 | 0.000 | 542 | 2.1 | 53.3% | 100 | -5 | 154.7 | 0.0 | 154.7 | 1.9 | 155 | ****** |

Kernel mxv_pfx_ballot:

| | | | | | R-Eff-% | | -L2-Cache-- | | ---DRAM---- | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| wp | ac | work | t/$\mu$s | I/op | SM eff | Ld | St | Rd $\theta$ | Wr $\theta$ | Rd $\theta$ | Wr $\theta$ | GB/s | Data BW Util- |
| 32 | 32 | 1.000 | 7498 | 2.9 | 74.2% | 100 | 49 | 100.7 | 183.2 | 100.7 | 89.5 | 190 | ******* |
| 32 | 32 | 0.750 | 6709 | 3.1 | 71.2% | 100 | 49 | 87.9 | 152.1 | 87.6 | 75.1 | 163 | ****** |
| 32 | 32 | 0.500 | 4784 | 3.3 | 66.9% | 100 | 49 | 88.2 | 143.2 | 87.9 | 70.3 | 158 | ****** |
| 32 | 32 | 0.250 | 2854 | 3.8 | 56.7% | 100 | 48 | 89.0 | 122.5 | 88.8 | 59.0 | 147 | ***** |
| 32 | 32 | 0.021 | 1557 | 6.3 | 14.7% | 100 | 36 | 64.6 | 25.4 | 64.6 | 9.8 | 72 | ** |
| 32 | 32 | 0.004 | 1355 | 7.3 | 3.6% | 100 | 13 | 65.8 | 15.7 | 65.8 | 2.8 | 66 | ** |
| 32 | 32 | 0.000 | 862 | 6.9 | 1.1% | 100 | 12 | 97.3 | 0.0 | 97.3 | 1.2 | 97 | *** |

Kernel mxv_pfx_prob1:

| | | | | | R-Eff-% | | -L2-Cache-- | | ---DRAM---- | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| wp | ac | work | t/$\mu$s | I/op | SM eff | Ld | St | Rd $\theta$ | Wr $\theta$ | Rd $\theta$ | Wr $\theta$ | GB/s | Data BW Util- |
| 32 | 32 | 1.000 | 7676 | 2.4 | 92.3% | 100 | 50 | 98.4 | 174.9 | 98.4 | 87.4 | 186 | ******* |
| 32 | 32 | 0.750 | 7093 | 2.5 | 91.5% | 100 | 50 | 82.8 | 141.9 | 82.8 | 71.0 | 154 | ****** |
| 32 | 32 | 0.500 | 5473 | 2.6 | 89.9% | 100 | 50 | 76.7 | 122.6 | 76.6 | 61.4 | 138 | ***** |
| 32 | 32 | 0.250 | 4829 | 2.8 | 85.6% | 100 | 50 | 52.1 | 69.5 | 52.1 | 34.9 | 87 | *** |
| 32 | 32 | 0.021 | 1167 | 3.3 | 58.2% | 100 | 50 | 84.3 | 24.7 | 84.3 | 13.0 | 97 | *** |
| 32 | 32 | 0.004 | 658 | 2.9 | 50.9% | 100 | 51 | 131.8 | 8.5 | 131.7 | 5.5 | 136 | ***** |
| 32 | 32 | 0.000 | 542 | 2.7 | 68.4% | 100 | 109 | 154.9 | 0.0 | 154.9 | 1.9 | 155 | ****** |

Kernel mxv:

| | | | | | | R-Eff-% | | -L2-Cache-- | | ---DRAM---- | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| wp | ac | work | t/$\mu$s | I/op | SM eff | Ld | St | Rd $\theta$ | Wr $\theta$ | Rd $\theta$ | Wr $\theta$ | GB/s | Data BW Util- |
| 8 | 8 | 1.000 | 8141 | 2.2 | 100.0% | 100 | 50 | 92.7 | 164.9 | 92.7 | 82.4 | 175 | ******* |
| 8 | 8 | 0.750 | 7028 | 2.9 | 81.4% | 100 | 54 | 83.6 | 132.0 | 83.6 | 71.7 | 155 | ****** |
| 8 | 8 | 0.500 | 5792 | 4.1 | 64.5% | 100 | 67 | 72.4 | 86.9 | 72.4 | 58.0 | 130 | ***** |
| 8 | 8 | 0.250 | 5179 | 7.1 | 52.0% | 100 | 86 | 48.6 | 37.5 | 48.6 | 32.5 | 81 | *** |
| 8 | 8 | 0.021 | 2746 | 11.8 | 48.4% | 100 | 100 | 35.8 | 5.3 | 35.8 | 5.6 | 41 | * |
| 8 | 8 | 0.004 | 1722 | 5.1 | 48.6% | 100 | 100 | 50.4 | 1.6 | 50.3 | 2.2 | 52 | ** |
| 8 | 8 | 0.000 | 1373 | 2.1 | 53.3% | 100 | -5 | 61.1 | 0.0 | 61.1 | 0.7 | 61 | ** |

Kernel mxv_pfx_ballot:

| | | | | | | R-Eff-% | | -L2-Cache-- | | ---DRAM---- | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| wp | ac | work | t/$\mu$s | I/op | SM eff | Ld | St | Rd $\theta$ | Wr $\theta$ | Rd $\theta$ | Wr $\theta$ | GB/s | Data BW Util- |
| 8 | 8 | 1.000 | 8212 | 2.9 | 73.2% | 100 | 46 | 91.9 | 178.8 | 91.9 | 81.7 | 174 | ******* |
| 8 | 8 | 0.750 | 7491 | 3.2 | 68.3% | 100 | 47 | 79.6 | 142.8 | 78.7 | 67.2 | 146 | ***** |
| 8 | 8 | 0.500 | 6356 | 3.5 | 63.0% | 100 | 46 | 67.4 | 115.1 | 66.7 | 52.9 | 119 | **** |
| 8 | 8 | 0.250 | 5717 | 4.1 | 51.4% | 100 | 42 | 45.6 | 69.7 | 45.2 | 29.5 | 73 | ** |
| 8 | 8 | 0.021 | 4524 | 8.2 | 8.3% | 100 | 14 | 23.6 | 22.2 | 23.6 | 3.4 | 25 | * |
| 8 | 8 | 0.004 | 3704 | 8.6 | 4.8% | 100 | 13 | 26.4 | 6.1 | 26.4 | 1.0 | 24 | |
| 8 | 8 | 0.000 | 2458 | 6.9 | 1.1% | 100 | 12 | 34.1 | 0.0 | 34.1 | 0.4 | 34 | * |

Kernel mxv_pfx_prob1:

| | | | | | | R-Eff-% | | -L2-Cache-- | | ---DRAM---- | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| wp | ac | work | t/$\mu$s | I/op | SM eff | Ld | St | Rd $\theta$ | Wr $\theta$ | Rd $\theta$ | Wr $\theta$ | GB/s | Data BW Util- |
| 8 | 8 | 1.000 | 7900 | 2.4 | 92.3% | 100 | 50 | 95.6 | 169.9 | 95.6 | 85.0 | 181 | ******* |
| 8 | 8 | 0.750 | 6976 | 2.5 | 91.5% | 100 | 50 | 84.2 | 144.3 | 84.2 | 72.2 | 156 | ****** |
| 8 | 8 | 0.500 | 5148 | 2.6 | 90.0% | 100 | 50 | 81.5 | 130.4 | 81.5 | 65.3 | 147 | ***** |
| 8 | 8 | 0.250 | 3446 | 2.8 | 85.7% | 100 | 50 | 73.0 | 97.4 | 73.0 | 48.9 | 122 | **** |
| 8 | 8 | 0.021 | 1762 | 3.3 | 58.6% | 100 | 50 | 55.8 | 16.3 | 55.8 | 8.6 | 64 | ** |
| 8 | 8 | 0.004 | 1480 | 2.9 | 52.4% | 100 | 50 | 58.6 | 3.8 | 58.6 | 2.5 | 60 | ** |
| 8 | 8 | 0.000 | 1374 | 2.7 | 43.1% | 100 | 120 | 61.1 | 0.0 | 61.1 | 0.7 | 61 | ** |

Problem 2: [50 pts] Hegde proposes UCNN, a CNN accelerator design that exploits repeated weights [2]. In UCNN the order in which inputs are read is determined by the weight values applied to them, enabling inputs to be summed first, and then multiplied by their common weight. This requires as few as one register for accumulating the sum, but requires each PE to buffer the entire input, or else a group of PEs to be fed parts of a long contiguous read. The `iiT` array maps an index (perhaps a simple loop iterator) to an input number.

In UCNN a PE, using `q=0,1,2,` as an iterator, computes `a += i[n][c][x+iiTx[m][c][q]][y+iiTy[m][c][q]]` and during this computation sets `o[n][m][x][y] += a * wiT[m][c][]; a = 0;` at group boundaries (details not shown).

Consider the following alternative design, CNNU. The *weight index table* array `wiT[M][C][I][J]` holds weight indices and the *weight value table* `wvT[M][C][U]` holds weight values, where $M$ is the number of output channels, $C$ is the number of input channels, $I$ and $J$ are the number of rows and columns in the filter, and $U$ is the maximum number of distinct weights per filter (per $m, c$ pair).

Instead of accessing `w[m][c][i][j]` to retrieve the weight for output channel $m$, input channel $c$, filter coordinate $i, j$, this system could access `wvT[m][c][ wiT[m][c][i][j] ]`.

A PE for an output-stationary dataflow would have $U$ registers, `a[U]`, for each output, these would be initialized to zero. Input and `wiT` values would be stored in or streamed to the PEs. The PE would compute `a[ wiT[m][c][i][j] ] += i[n][c][x+i][y+j]` for each `I` and `J`. Finally, the weights would be applied: `o[n][m][x][y] += a[u] * wvT[m][u]` .

Here are some important differences: A PE would need $U$ to $GU$ accumulators instead of 1 to $G$. Each entry in the `wiT` would require only $\lceil \lg U \rceil$ bits.

Note: CNNU was made up for this exam. However, it's a fairly obvious way to exploit weight re-use and so the idea probably has been described already in the literature.

Answer questions on the next page, which are about UCNN as described by Hegde and about CNNU as described here.

(*a*) Compare the amount of storage needed for weights and the indirection tables (before loading them onto the chip) between UCNN and CNNU. Consider systems designed for a few unique weights and lots of unique weights. Make up values for $U$ and $IJ$ (number of weights per filter) for your answer.

(*b*) It seems as though with UCNN each PE must use a different input element than its neighbors. This would require each PE to store an entire input channel, or the global buffer would have to provide a different value to each PE (which would be costly), or, as the paper suggests, groups of PEs share a banked memory. But in all of these situations each input value is read from a medium-sized to large memory and used once.

In an ordinary output-stationary CNN one input value can be used to compute multiple output channels, so long as the PE has storage for those channels. Explain why that can't be done using UCNN but is still possible using CNNU.

(*c*) How easy would it be to apply UCNN to the row-stationary dataflow of Chen [1]. What sort of shapes would work well and poorly?

(*d*) Consider the convolutional layer shapes for AlexNet as tabulated in [1] in Table II. Pay attention to the size of the filter and the number of channels. What are the implications of the small filter sizes? How might UCNN be modified to accommodate them?

**References:**


[1] Chen, Y.-H., Emer, J., and Sze, V. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2016), ISCA '16, IEEE Press, pp. 367–379. https://doi.org/10.1109/ISCA.2016.40.

[2] Hegde, K., Yu, J., Agrawal, R., Yan, M., Pellauer, M., and Fletcher, C. W. UCNN: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2018), ISCA '18, IEEE Press, pp. 674–687. https://doi.org/10.1109/ISCA.2018.00062.