Contents

1 Abstract

To explore the relative merits between single-node computer power in depth versus multi-node compute power in breadth, the author proposed an experiment comparing the performance and efficiency of Cannon's algorithm[17, 15] for the matrix multiplication problem $C = A \times B$ between a single-node OpenMP[27] implementation utilizing accelerators and a multi-node MPI[25] implementation spread across a cluster.

1.1 Data

For the sake of time, only the C-stationary case for a limited number of matrix sizes (256x256, 1024x1024, 4096x4096, 16384x16384, 256x1024, 256x4096,

256x16384, 16384x256, 16384x256, 4096x256, 1024x256) were considered. The matrices were to be filled with random numbers for the performance trials. Validation trials were to be made through specially devised matrices A and B such that each cell of the product matrix C would have a unique value.

1.2 Resources

The OpenMP trials were to be executed on the GPU nodes on BRIDGES[1] and/or the KNL nodes on STAMPEDE2, whereas the MPI trials would be executed on the appropriate non-accelerated nodes on BRIDGES.

1.3 Efficiency

The efficiency of the MPI trials were gauged against a strong-scaling roofline model based on the physical characteristics of a single non-GPU node on BRIDGES. The processor in this case was the 28-core, 2.30 GHz Intel E5-2695, which is theoretically capable of 2.30 GHz * 28 cores * 4 SIMD instructions/cycle (AVX256) = 257.6 GFLOPs/s/node.

The efficiency of the OpenMP trials were gauged against a roofline model based on the physical characteristics of the accelerator. The NVIDIA P100 is theoretically capable of 9.3 SP TFLOPs/s/card[26], whereas the NVIDIA K80 is theoretically capable of 8.74 TFLOPs/s/card[30]. In the event that KNL nodes on STAMPEDE2 were utilized, the theoretical roofline would be considered as 1.4 GHz * 68 cores * 8 SIMD instructions/cycle (AVX512) = 761.6 GFLOPs/s/node[32][2].

The matrix multiplication functions provided with the Intel Math Kernel Library (MKL) were consulted as a reference for empirically achievable single-node performance either without GPU acceleration or in the case of KNL nodes while the matrix multiplication sample provided by the CUDA toolkit utilizing CUBLAS functions was consulted as a reference for empirically achievable single-node performance with GPU acceleration.

1.4 Hypotheses

It was predicted that the accelerated OpenMP solution would outperform the MPI solution for smaller matrix sizes until reaching an inflection point where the contention between resources within the node is a greater bottleneck than the communication overhead across multiple nodes. This inflection point would largely depend on the architecture and layout of the accelerator used, i.e., the dimensions of the various hierarchical groupings of its compute resources.

[1] If the GPU nodes on BRIDGES did not support OpenMP offloading to the P100 and K80 GPUs, the acceleration would be done via OpenACC[33], or, as a last resort, CUDA.

[2] While KNL supports 4 threads/core, only 1 was considered here as performance may degrade over shared resources.

2 Validation

Before commencing on the actual implementation of the matrix multiplication programs, it was determined that the design of the validation trials would take first priority so as to catch errors in implementation earlier rather than later. The validation mechanism itself needed to operate under several exacting constraints. It has been already mentioned that the $m \times n$ product matrix C in validation mode would need to house a unique value in each cell once calculations had finished. This was to ensure that every dot product of each row in $m \times q$ matrix A and each column in $q \times n$ matrix B did not suffer corruption from any other dot products or other implementation errors by providing a quick reference that could be used as a sanity check.

Alacrity in the validation implementation was key. If the validation required additional extended calculations to replicate the results serially, the time to find and correct errors could become prohibitively slow, especially as the matrix dimensions m, q, and n increased. Ideally, the validation would be a constant time function f based on the row and column index of the cell in question, i.e., $C_{i,j} = f(i, j)$.

The under-the-hood 1D row-major ordering and indexing of 2D matrices in the C programming language provided the requisite inspiration. The cell of every 2D matrix in C is accessed implicitly by the following formula: $iq+j$, where i is the row index, j is the column index, and n is the number of columns[18, p. 113]. Initial attempts sough to find some function $f(i, j) = \sum_{k=1}^{N} A_{i,k} \cdot B_{k,j}$ such that the matrix A independently provided the i component while the matrix B independently provided the j component.

After wrestling with the algebra for some time, a simpler solution was devised. Each cell of the matrix A would be populated with its row index i. Each cell of the matrix B would be populated with the constant 1. Each cell of the product matrix C would be pre-populated with its column index j instead of the customary constant 0. In this way, the equation $C = A \times B$ became $C = A \times B + C_0$ where performance trials would start with $C_0 = \mathbf{0}$ and validation trials would start with $C_0 = [\mathbf{0}\ \mathbf{1}\ \ldots\ \mathbf{m}]$.

While this satisfies the desired formula $in + j$ per cell, it did not provide a checksum against column drift since B is simply a homogeneous field of ones. The basic principle could be improved as follows. Let each cell of the matrix A be populated with the value $i + 1$. Let each cell of the matrix B be populated with the value j. If a basis of $C_0 = \mathbf{0}$ were assumed, each cell of the product matrix C would contain the value $q(i + 1)j$, which expands to $qij + qj$. By setting the basis matrix C_0 so that each cell contained $-((q - 1)j + (j - 1)iq)$, each cell of the product matrix C would then contain the desired value $iq + j$.

This is a relatively safe method for obtaining the desired value since there are no division operations required.

The problem with this latest refinement and even all the previous refinements is that the resultant values did not fit within the range of precise integral values available in single-precision floating point, i.e., $[-2^{24},2^{24}]$[29], given the target matrix sizes. The validation method needed to be able to precisely support values as high as $16383 \times 16383 + 16382 \times 16383 \times 16384$, or something on the order of 2^{42}. Several attempts to compress the range by reducing i and j failed to produce the desired results and only complicated the requisite basis matrix C_0 with no guarantee of precision fidelity.

Furthermore, the previous equation made an untenable assumption in view of the proposed target matrix sizes. The inner dimension q might not be equal to the final column dimension n of the matrix C. Such an invariant only holds if the matrices A and B are square.

3 Empirical Rooflines

In order to get a gauge for the current state of the art as regards matrix multiplication, it was necessary to develop a couple of test harnesses. These harnesses could provide a reasonable estimate of what performance and efficiency would be achievable within the bounds of the peak theoretical rooflines. Because of the difference in architectures between the scale-up and scale-out cases, at least two harnesses would need to be developed.

3.1 Scale-Up

For the GPU offloading or scale-up case, the NVIDIA CUBLAS library was selected as the empirical roofline model. It is assumed that the development staff at NVIDIA has a more intimate knowledge of the NVIDIA P100 and NVIDIA K80 platforms than the general public and therefore should be better equipped to maximize resource utilization for those architectures.

The `matrixMulCUBLAS` sample source provided with CUDA Toolkit 8.0 served as the basis for the CUBLAS test harness. Some minor modifications were made to the command-line arguments to support a wider array of target matrix sizes as the original source hardcoded the dimensions based on a single multiplier. The `Makefile` was also modified to make the entire harness self-contained, including some header files common to most of the sample programs provided with the CUDA toolkit.

To run the scale-up harness, one would type the following command:

`./matrixMulCUBLAS m=`m `q=`q `n=`n `validation=[0|1]`

The original source validated the CUBLAS results by calculating a reference solution on the host. Given the target matrix sizes involved, this could have become prohibitively expensive in terms of time cost, especially in early stage development on the author's local laptop machine. This validation was made optional through the `validation` command-line argument. The default was to forgo validation, but validation of a sort could be re-enabled by adding `validation=1` to the command-line call. In the end, the only validation that was done was to seed the matrices A and B as homogeneous fields of ones, i.e., $A = \mathbf{1}$ and $B = \mathbf{1}$. The validation simply checked that every cell of the product matrix C was set to q.

The arguments `m`, `q`, and `n` could be any arbitrary integer, and no check was done for alignment against the CUDA card except what may have existed in the original source. Since the target matrix sizes for the experiment were limited to powers of two (2), this was deemed an acceptable risk.

The output of the program had a fair amount of helpful human-readable information that was piped to `stderr`. Most of the `stderr` output had its origins in the original source provided with the CUDA toolkit. On the other hand, output to `stdout` was reserved for machine-readable CSV text. By convention, the output consisted of the following columns:

1. m, the row dimension of A and C
2. q, the column dimension of A and the row dimension of B
3. n, the column dimension of B and C
4. r, the achieved rate (gigaflops/sec)
5. s, the number of iterations
6. f, the number of flops per iteration (flops)
7. t_{mult}, the average time spent per iteration on multiplication (ms)
8. t_{comm}, the time spent in host-to-device or device-to-host transfers (ms)

3.2 Scale-Out

For the unaccelerated MPI or scale-out case, the Intel MKL libraries and their concomitant BLAS3 functions were selected as the empirical roofline model. Since all of the scale-out cases would be run on nodes with Intel processors, it

was assumed that the development staff at Intel had a more intimate knowledge of the processors involved than the general public and therefore should be better equipped to maximize resource utilization on these architectures.

There was a slight difference in the methodology involved here from that employed in the scale-up case. Whereas the scale-up case looked at resource utilization holistically with regard to the accelerator, the harness for the scale-out case would evaluate the single-core performance of the MKL BLAS3 function `cblas_sgemm`[6], and the roofline would be extrapolated along a strong-scaling model so that a scale-out case running with p processors would be measured against the single-core rate multiplied by p to determine its efficiency relative to MKL.

The sample code provided by Intel for `cblas_sgemm`[7] was consulted as a reference for the scale-out single-core basis harness to validate correctness of the implementation, particularly the call to `cblas_sgemm`, but the API documentation served as the primary resource that informed the implementation not only of the MKL library usage but the devising of command-line options as well. In fact, the command-line options of the scale-up harness were altered to match those of the scale-out harness for legibility and ease of batch scripting later.

To run the scale-out harness, one would type the following command:

`./mkl_cblas_sgemm -m` m `-q` q `-n` n `validation=[0|1]`

The arguments provided to `-m`, `-q`, and `-n` could be any arbitrary integer.

Output to `stdout` was reserved for machine-readable CSV text. By convention, the output consisted of the following columns:

1. m, the row dimension of A and C
2. q, the column dimension of A and the row dimension of B
3. n, the column dimension of B and C
4. r, the achieved rate (gigaflops/sec)
5. s, the number of iterations
6. f, the number of flops per iteration (flops)
7. t_{mult}, the average time spent per iteration on multiplication (ms)

4 Implementation

4.1 Scale-Up

Initial development on the scale-up implementation of Cannon's algorithm began with OpenMP. Some difficulties were encountered with regard to the offset math required for setting up a non-square 2D grid which delayed development of the parallelization. Once the problem was reduced to the square target matrix sizes, it was realized that part of the problem was the failure to enforce the rigidity of the grid dimensions coupled with a misreading of the algorithm description with regard to the initial shearing of phase 1.

An additional hurdle was encountered when it became clear that OpenMP GPU offloading was not enabled with either gcc or icc on the intended BRIDGES computing cluster. Reading the GPU user guide for BRIDGES[3] made it clear that the preferred and supported means of `#pragma`-centric acceleration was OpenACC with the PGI compiler. A quick perusal of the GCC documentation for OpenACC support[14] revealed a series of known issues that made GCC infeasible for use, even with the latest 7.2 release. Namely, the lack of support for host fallback and nested parallel loop support eliminated GCC from consideration.

Initially, the OpenACC implementation was thought to be functional if not performant. However, there were additional logic errors in the offset math that needed to be corrected along with further modifications to take advantage of the GPU acceleration, primarily in the block multiplication phase. First attempts with the pgcc compiler only added serial loops on the GPU because of complex loop variable dependencies.

It was determined that a more rigorous examination of the parallelization was required. The ordering of the loop variables needed changing and a better understanding of OpenACC directives was sought. After reviewing some tutorials on OpenACC parallelization techniques[9, 8], some experiments with loop ordering were attempted and explicit `#pragma acc loop independent` directives were added in the more deeply nested loops. The alleged conflicts with `dA` and `dB` disappeared, but the outermost loops were still sequential. It was not until violating one of the most widely disseminated best practices, namely the avoidance of pointer arithmetic, that all the loops became parallelizable. After a quick verification check that the results were valid for a small case, the smallest target matrix size was tested under a nvprof wrapper with the following results:

```
michael@aphelion ~/cosc6365/final $ nvprof ./cannon-su-acc -p 16 -m 256 -q 256 -n 256
==12882== NVPROF is profiling process 12882, command: ./cannon-su-acc -p 16 -m 256 -q 256 -n 256
16,256,256,256,1,33554432.00,0.00,1319.012,82.073
==12882== Profiling application: ./cannon-su-acc -p 16 -m 256 -q 256 -n 256
==12882== Profiling result:
```

```
Time(%)      Time     Calls       Avg       Min       Max  Name
99.00%  1.31888s         4  329.72ms  326.65ms  337.61ms  main_356_gpu
0.61%  8.1765ms      2049  3.9900us  3.6800us  169.38us  [CUDA memcpy HtoD]
0.37%  4.9208ms      5144     956ns     896ns  10.432us  [CUDA memcpy DtoD]
0.01%  160.00us         1  160.00us  160.00us  160.00us  [CUDA memcpy DtoH]

==12882== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
85.39%  1.33983s      2054  652.30us  4.1210us  337.62ms  cuStreamSynchronize
6.29%  98.630ms         1  98.630ms  98.630ms  98.630ms  cuDevicePrimaryCtxRetain
4.16%  65.316ms         1  65.316ms  65.316ms  65.316ms  cuDevicePrimaryCtxRelease
1.86%  29.147ms      5144  5.6660us  5.0490us  325.27us  cuMemcpyDtoDAsync
1.03%  16.215ms         1  16.215ms  16.215ms  16.215ms  cuMemHostAlloc
0.59%  9.2858ms         1  9.2858ms  9.2858ms  9.2858ms  cuMemFreeHost
0.41%  6.4971ms      2049  3.1700us  2.9330us  23.480us  cuMemcpyHtoDAsync
0.17%  2.7347ms     10288     265ns     211ns  2.7000us  cuPointerGetAttributes
0.04%  695.98us         4  174.00us  3.9990us  241.37us  cuMemAlloc
0.03%  409.59us         1  409.59us  409.59us  409.59us  cuMemAllocHost
0.01%  149.14us         1  149.14us  149.14us  149.14us  cuModuleLoadData
0.00%  32.707us         4  8.1760us  6.7910us  10.511us  cuLaunchKernel
0.00%  21.719us         1  21.719us  21.719us  21.719us  cuStreamCreate
0.00%  20.038us         1  20.038us  20.038us  20.038us  cuMemcpyDtoHAsync
0.00%  5.6230us         2  2.8110us     454ns  5.1690us  cuEventCreate
0.00%  2.5610us         1  2.5610us  2.5610us  2.5610us  cuEventRecord
0.00%  2.2920us         3     764ns     163ns  1.7650us  cuDeviceGetCount
0.00%  1.8190us         3     606ns     293ns  1.0710us  cuCtxSetCurrent
0.00%  1.3030us         3     434ns     272ns     731ns  cuDeviceGet
0.00%  1.1740us         1  1.1740us  1.1740us  1.1740us  cuModuleGetFunction
0.00%  1.1310us         4     282ns     250ns     338ns  cuDeviceGetAttribute
0.00%  1.0130us         1  1.0130us  1.0130us  1.0130us  cuEventSynchronize
0.00%     317ns         1     317ns     317ns     317ns  cuMemFree
0.00%     292ns         1     292ns     292ns     292ns  cuCtxGetCurrent
0.00%     289ns         1     289ns     289ns     289ns  cuDeviceComputeCapability

==12882== OpenACC (excl):
Time(%)      Time     Calls       Avg       Min       Max  Name
91.10%  1.31913s         5  263.83ms  210.04us  337.63ms  acc_wait@byteswap.h:356
6.87%  99.469ms         1  99.469ms  99.469ms  99.469ms  acc_device_init
1.50%  21.778ms      2049  10.628us  4.7710us  171.59us  acc_wait
0.52%  7.5036ms      2049  3.6620us  3.4070us  29.945us  acc_enqueue_upload
0.00%  44.429us         4  11.107us  9.2100us  14.338us  acc_enqueue_launch@byteswap.h:356 (main_356_gpu)
0.00%  34.080us         1  34.080us  34.080us  34.080us  acc_enqueue_download@byteswap.h:356
0.00%  29.343us         4  7.3350us  5.5510us  12.177us  acc_compute_construct@byteswap.h:356
0.00%       0ns         3       0ns       0ns       0ns  acc_delete@(OpenACC API):1
0.00%       0ns         3       0ns       0ns       0ns  acc_alloc@(OpenACC API):1
```

The modifications garnered roughly 4 times better performance than prior attempts. As evidenced by the nvprof output, the bulk of time was being spent in `acc_wait`, an implicit synchronization emplaced to protect the contents of `dC`, but at least the multiplication was finally out of the hands of the host processor. The question then became how to reduce the rather significant amount of context swapping.

A quick experiment in replacing the `#pragma acc parallel` directive with `#pragma acc kernels` immediately doubled the performance. Another brief experiment in reverting to ijk multiplication and leveraging the `#pragma acc loop reduction` directive[36, p. 2] revealed a disappointing truth: the program ran fastest when p was set to 1, i.e., no communication between blocks. The attempt to save on memory costs and, in theory, time, by allocating the device copies of the matrices A, B, and C as one large block each seemed to defeat the expected benefits of Cannon's algorithm and its promise of allocation of relevant data close to the processors involved.

Further experimentation with precalculating offsets yielded promising results. By moving to an $ixykj$ loop model and precalculating the parts of the offsets pertaining to i, x, and y in the y loop, performance was greatly improved.

```
michael@aphelion ~/cosc6365/final $ nvprof ./cannon-su-acc -p 16 -m 256 -q 256 -n 256
==8811== NVPROF is profiling process 8811, command: ./cannon-su-acc -p 16 -m 256 -q 256 -n 256
16,256,256,256,1,33554432.00,0.00,136.267,81.453
==8811== Profiling application: ./cannon-su-acc -p 16 -m 256 -q 256 -n 256
==8811== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
91.06%  136.17ms         4  34.042ms  31.990ms  35.053ms  main_344_gpu
5.38%  8.0485ms      2049  3.9280us  3.6800us  169.38us  [CUDA memcpy HtoD]
3.45%  5.1647ms      5144  1.0040us     896ns  14.912us  [CUDA memcpy DtoD]
0.11%  160.80us         1  160.80us  160.80us  160.80us  [CUDA memcpy DtoH]

==8811== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
40.09%  156.51ms      2054  76.199us  3.9950us  35.060ms  cuStreamSynchronize
27.05%  105.60ms         1  105.60ms  105.60ms  105.60ms  cuDevicePrimaryCtxRetain
16.28%  63.552ms         1  63.552ms  63.552ms  63.552ms  cuDevicePrimaryCtxRelease
7.37%  28.755ms      5144  5.5900us  5.0030us  363.46us  cuMemcpyDtoDAsync
4.05%  15.824ms         1  15.824ms  15.824ms  15.824ms  cuMemHostAlloc
2.42%  9.4366ms         1  9.4366ms  9.4366ms  9.4366ms  cuMemFreeHost
1.69%  6.6098ms      2049  3.2250us  2.9770us  23.551us  cuMemcpyHtoDAsync
0.70%  2.7276ms     10288     265ns     208ns  13.388us  cuPointerGetAttributes
0.18%  719.74us         4  179.93us  4.3360us  264.30us  cuMemAlloc
0.10%  405.40us         1  405.40us  405.40us  405.40us  cuMemAllocHost
0.04%  158.18us         1  158.18us  158.18us  158.18us  cuModuleLoadData
0.01%  44.311us         1  44.311us  44.311us  44.311us  cuStreamCreate
0.01%  26.994us         4  6.7480us  5.5120us  9.2010us  cuLaunchKernel
0.01%  19.681us         3  6.5600us  1.0700us  17.235us  cuDeviceGet
0.00%  11.049us         1  11.049us  11.049us  11.049us  cuMemcpyDtoHAsync
0.00%  5.9280us         3  1.9760us     486ns  3.7500us  cuDeviceGetCount
0.00%  3.9700us         4     992ns     821ns  1.4170us  cuDeviceGetAttribute
0.00%  2.9060us         2  1.4530us     469ns  2.4370us  cuEventCreate
0.00%  2.1130us         1  2.1130us  2.1130us  2.1130us  cuEventRecord
0.00%  1.9380us         3     646ns     257ns  1.2730us  cuCtxSetCurrent
0.00%  1.2690us         1  1.2690us  1.2690us  1.2690us  cuModuleGetFunction
0.00%  1.0160us         1  1.0160us  1.0160us  1.0160us  cuEventSynchronize
0.00%     941ns         1     941ns     941ns     941ns  cuDeviceComputeCapability
0.00%     903ns         1     903ns     903ns     903ns  cuCtxGetCurrent
0.00%     429ns         1     429ns     429ns     429ns  cuMemFree
```

```
==8811== OpenACC (excl):
Time(%)      Time     Calls       Avg       Min       Max  Name
50.16%  136.42ms         5  27.284ms  223.56us  35.061ms  acc_wait@byteswap.h:344
39.15%  106.47ms         1  106.47ms  106.47ms  106.47ms  acc_device_init
7.84%  21.312ms      2049  10.401us  4.8300us  169.38us  acc_wait
2.83%  7.7034ms      2049  3.7590us  3.5020us  29.916us  acc_enqueue_upload
0.01%  35.966us         4  8.9910us  7.1280us  12.875us  acc_enqueue_launch@byteswap.h:344 (main_344_gpu)
0.01%  20.712us         4  5.1780us  2.8470us  10.060us  acc_compute_construct@byteswap.h:339
0.01%  20.466us         1  20.466us  20.466us  20.466us  acc_enqueue_download@byteswap.h:344
0.00%       0ns         3       0ns       0ns       0ns  acc_delete@(OpenACC API):1
0.00%       0ns         3       0ns       0ns       0ns  acc_alloc@(OpenACC API):1
```

Under this $ixykj$ regimen, the `#pragma acc loop reduction` directive around the k loop was replaced with a `#pragma acc loop independent` directive since perfunctory tests with it revealed a decrease in performance when attempting an $ixyjk$ reduction model of parallelism. It is questionable whether the technique and its loop arrangement can be called strictly Cannon's algorithm, though one might envision it as one where all the individual processors process each i-component simultaneously if not completely independently. It is with this version that final tests runs were executed.

To run this scale-up implementation of Cannon's algorithm, one would type the following command:

`./cannon-su -p` p `-m` m `-q` q `-n` n

By convention, the output consisted of the following columns:

1. p, the number of processors in the processor grid
2. m, the row dimension of A and C
3. q, the column dimension of A and the row dimension of B
4. n, the column dimension of B and C
5. s, the number of trials
6. e, the particular trial being measured
7. f, the number of flops per iteration (flops)
8. r, the achieved rate (gigaflops/sec)
9. t_{mult}, the average time spent per iteration on multiplication (ms)
10. t_{comm}, the time spent in host-to-device or device-to-host transfers (ms)

4.2 Scale-Out

Development of the scale-out implementation borrowed the core of the scale-up OpenACC implementation and was ported to use the MPI library functions. This port employed the RMA communication mechanism with Post-Start-Complete-Wait synchronization[25, pp. 456-463] since the cyclic rotation between MPI processes must be marshalled in such a way that the present data is not overwritten.

Array Distribution During program startup, the root process $P_{0,0}$ filled the arrays A, B, and C and issued a `MPI_Win_post` call to `MPI_COMM_WORLD` to notify the other processes and itself that the arrays were ready for retrieval. Each process $P_{x,y}$ issued a `MPI_Get` request to the root process $P_{0,0}$ for the properly sheared 2D subarrays $A_{x,(y+x)\ mod\ c}$ and $B_{(x+y)\ mod\ b,y}$ along with the stationary 2D subarray $C_{x,y}$. These were stored in individual process memory and opened via another RMA window so that the requisite neighbor processes could access them during the cyclic rotation phases.

Cyclic Rotation Within each process, the memory allocated for the A and B subarrays was actually twice that required. This was to enable whole-block copying between neighbor processes. The additional space served as a send buffer during cyclic rotation and prevented corruption. Each process $P_{x,y}$ copied the entirety of the A (B) submatrix and then issued a `MPI_Win_post` call for the corresponding window to a group comprised of only its east (south) neighbor so that it could receive the other's copy of the corresponding submatrix. Immediately after the `MPI_Post` calls were sent, the process $P_{x,y}$ then issued a `MPI_Win_Start` call to its west (north) neighbor so that it could send its prior submatrix A (B) from the ghost block. Once that neighbor has posted its readiness, the process $P_{x,y}$ issued a `MPI_Put` to deliver the data. The transaction was finalized with a call to `MPI_Win_complete` on the corresponding window, and then the process issued a `MPI_Win_wait` to block until it had assurance that its supplier had finished the incoming transaction before proceeding to the multiplication phase.

Local Multiplication The local multiplication was implemented as a simple *ikj* multiplication on the local process block. Some attempts were made to optimize it with OpenMP, but the parallelizations turned out to be worse than the serial code in local development tests. In order to maximize the number of cores that could be devoted to the outward scaling with the minimum amount of node allocations, it was determined in the end to forgo any thread parallelization within each MPI process. After each local multiplication was finished, the process $P_{x,y}$ looped until it has processed the block row x and block column y at most once.

Collection When all the work had finished, the various processes reported back to the root process via individual `MPI_Put` calls to load balance the com-

munication. Future work might include investigating whether a more complex call like `MPI_Gather` would optimize the implementation.

To run this scale-out implementation of Cannon's algorithm, one would type the following command:

`mpirun -np` p `./cannon-so -m` m `-q` q `-n` n

By convention, the output consisted of the following columns:

1. p, the number of processors in the processor grid
2. m, the row dimension of A and C
3. q, the column dimension of A and the row dimension of B
4. n, the column dimension of B and C
5. s, the number of trials
6. e, the particular trial being measured
7. f, the number of flops per iteration (flops)
8. r, the achieved rate (gigaflops/sec)
9. t_{mult}, the average time spent per iteration on multiplication (ms)
10. t_{comm}, the time spent in host-to-device or device-to-host transfers (ms)

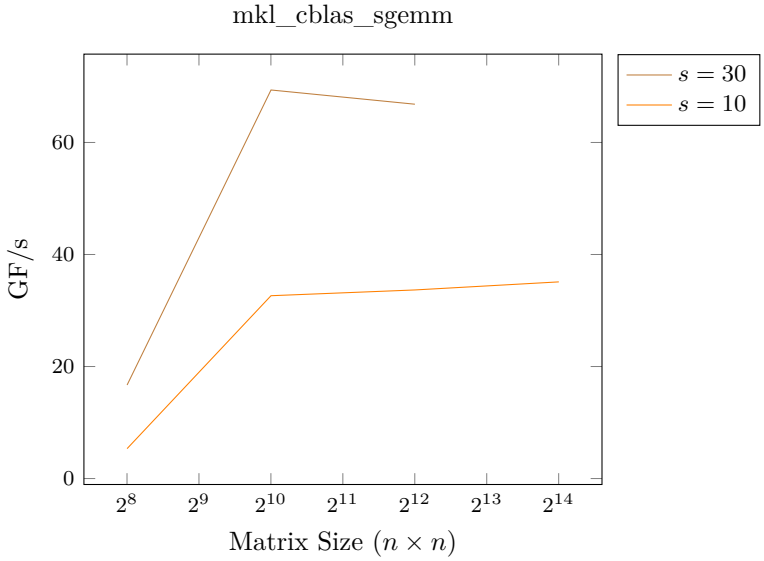
4.3 Data

It should be noted at this point that the test runs were not executed with random numbers as initially planned. Due to an oversight by the author, the performance trials for both the scale-up and scale-out versions were executed with each prospective block of the matrices filled with a block identifier which ran in sequence according to the row-major offset ordering of the C-programming language if one envisioned the matrix partitioned into a 2D array of blocks based on the number of processes. The refactoring to populate the matrices A and B random numbers was simply forgotten before trials began in earnest. The effect should be marginal but might be noticed in certain cases if the FPUs handling the multiplication have hardware optimization for multiplying certain numbers, for instance, powers of two.

5 Results

5.1 mkl_cblas_sgemm

The graph below depicts two sets of runs on the BRIDGES cluster with the `mkl_cblas_sgemm` harness. The first run consisted of 30 trials whereas the second run only consisted of 10 trials.

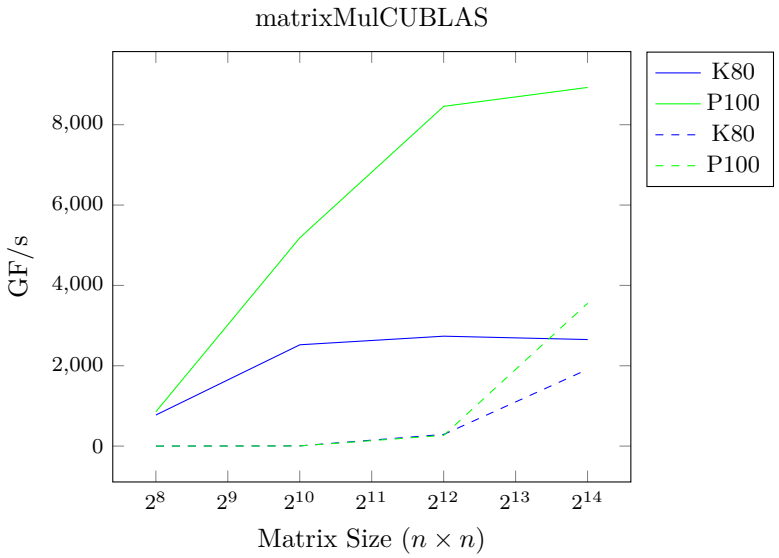


Curiously, the first run exhibited on average approximately double the performance of the second run. The author has no ready explanation for this since the test was executed on the RM partition with a dedicated node, so there should not have been any bleed from other jobs. It may be worth investigating in the future with a greater sample set, although the time involved would be considerable.

On grounds of statistical significance, the first run is potentially the better option to serve as a gauge, but its incompleteness with regard to the $m = q = n = 16384$ case makes it less viable. For the rest of this report, the numbers from the second run of ten trials are used for comparison for the sake of completeness across all the target matrix dimensions.

5.2 matrixMulCUBLAS

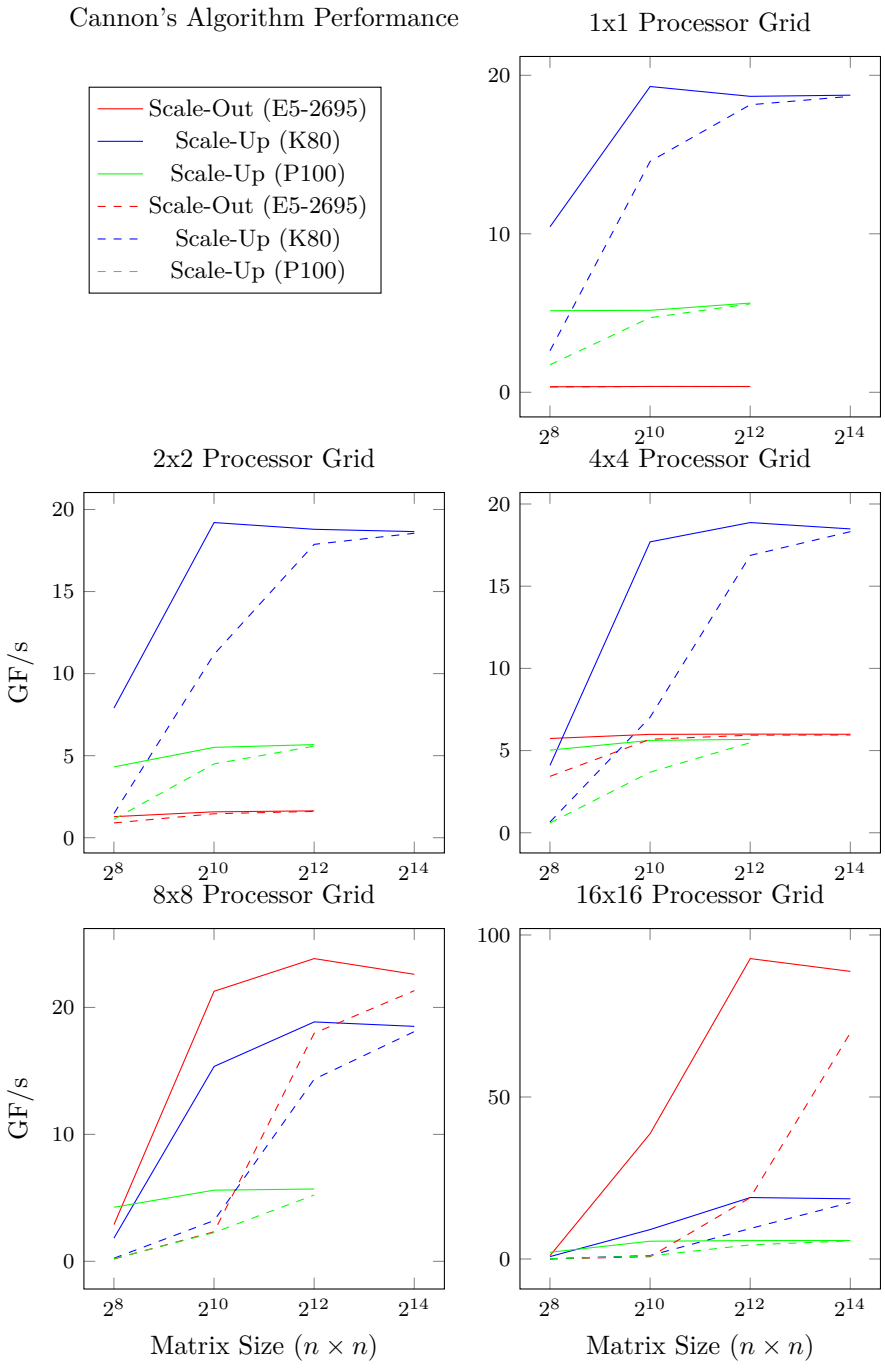
The graph below depicts runs on K80 and P100 nodes of the BRIDGES cluster with the `matrixMulCUBLAS` harness. The dashed lines include the time for device-to-host and host-to-device transfers in the GF/s calculation.



On the surface, the CUBLAS implementation achieves nearly peak TPP using NVIDIA's own methodology of calculating the arithmetic rate using only the multiplication time, particularly in the case of the P100. However, the entire system for dispatching and receiving the calculation results is not comprised solely of the accelerator card, and the significant time cost incurred by transferring the data to the card and back to the host should be reflected in the performance measurements, particularly when comparing against host-only solutions such as `mkl_cblas_sgemm`.

5.3 Cannon's Algorithm

Performance The following set of graphs depicts the performance of Cannon's Algorithm in both the scale-up and scale-out variants. Each graph tile represents the performance for a given square processor grid from 1 to 256 logical processors. The solid lines represent the arithmetic rate calculated only against t_{mult}, the time for multiplication, while the dashed lines represent the arithmetic rate calculated by including t_{comm} as synchronization delay and therefore as a legitimate damper on actual throughput.



For the scale-up variant, the raw multiplication arithmetic rate appears not to

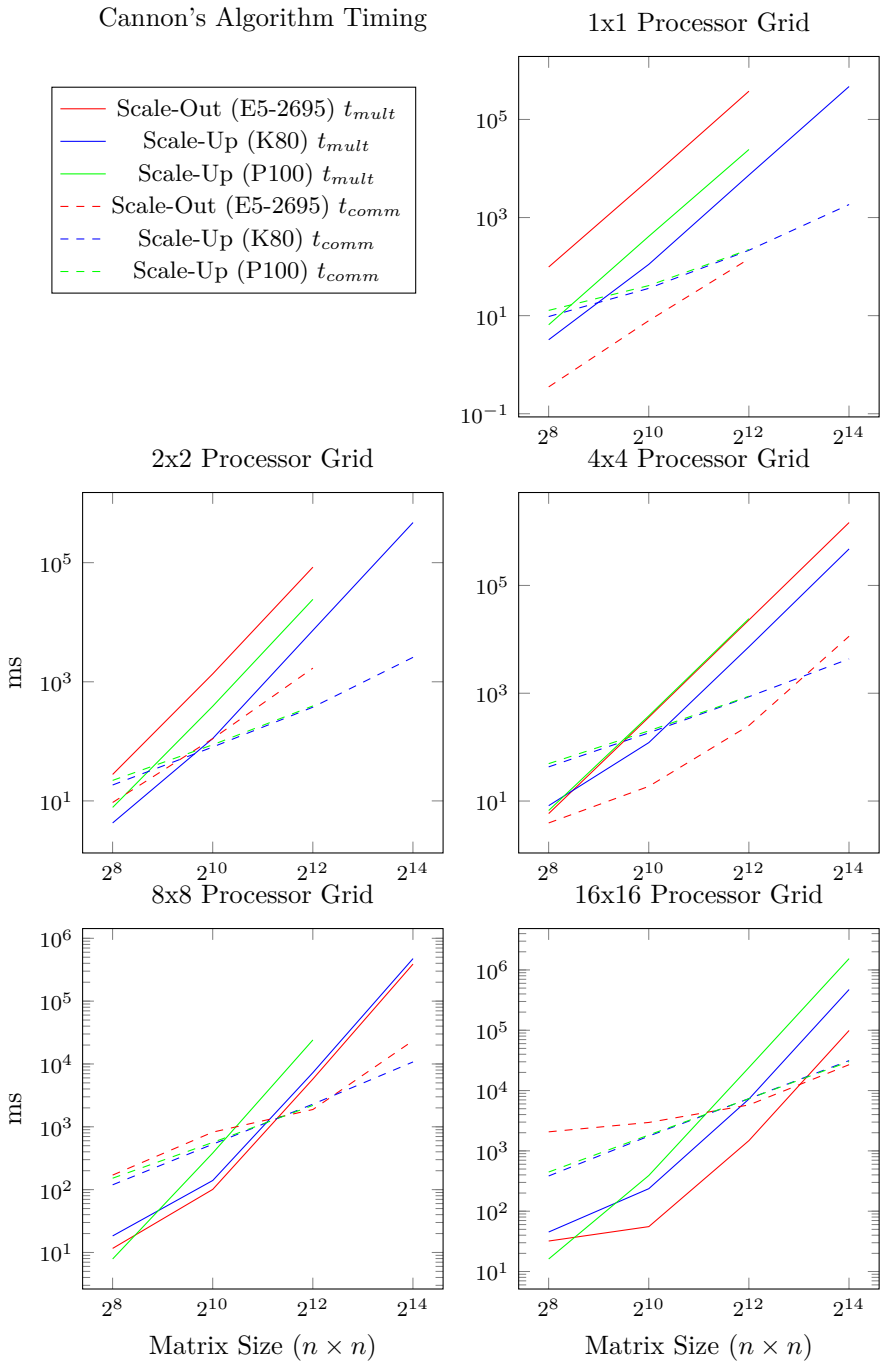
vary with changes to the processor grid. When the communication time between elements of the grid is factored in, the performance appears to degrade as the element count of the processor grid increases. This suggests that the overhead of moving pieces of the matrices around is not worth the reduction in size of the submatrices when the hardware is already so close together. Contrasting the performance against that of `matrixMulCUBLAS` inexorably leads one to the conclusion that Cannon's algorithm is not at all suited for running within the confines of a single GPU accelerator.

For the scale-out variant, the raw multiplication arithmetic increases proportional to the element count of the processor grid. There are fairly substantial gains on the part of the scale-out variant from the single-processor case to the 256 processor 16x16 grid relative to itself. As predicted, the scale-up variant handily outperforms the scale-out variant for smaller matrix sizes, but the scale-out variant come into its own when it has a large processor grid at its disposal, and this shift is most visible in the largest matrix sizes.

It was difficult to tell whether the partial validation of the initial hypothesis is entirely due to the factors that were considered at the onset or if a part may be played by the poverty of the scale-up implementation. With only a coarse window into the GPU hardware through the OpenACC directives and routines, the number of streaming processors actually doing the work in the single processor grid is unknown without further investigation. The performance results suggest that the GPU is effectively running the same amount of processors regardless of the logical processor grid size, probably as a result of the refinements to the OpenACC directives. In some sense, the attempt at implementing Cannon's algorithm on the GPU through OpenACC only added communication overhead with every increase in grid size.

That being said, there is definitely an upper limit on the amount of matrix that can be stored on a given GPU. While in theory the scale-out variant can keep scaling indefinitely as long as there are nodes and network connections between them, its primary limit is the fact that multiple processors cannot share a single matrix cell. Conversely, the developer is forced at some point with GPUs to either scale out to multiple GPUs or pipeline chunks of the matrix through the single GPU if the matrix size exceeds its physical capacity.

Timing As an alternate view into the results from the test runs of the scale-up and scale-out variants, the following graph depicts the timings that inform the previous performance graphs.



It may seem incongruous that a line for t_{comm} appears in the 1x1 scale-out

case, but even for this case, the root process makes copies of A, B, and C which it summarily gives itself to work from and dutifully copies back C. No special logic was put in place to avoid this overhead.

As the number of processors on the grid increases, the communication time begins to dominate until the matrix size increases to a point so that the amount of multiplication compares to the amount of data movement. The high cost of data movement becomes readily apparent when one considers the graphs and how many FLOPs are required to match or exceed the cost of transferring data. This perfectly encapsulates the Achilles' heel of Cannon's algorithm when it comes to performance. If one cannot minimize the constant communication cost, the algorithm will slow to a crawl.

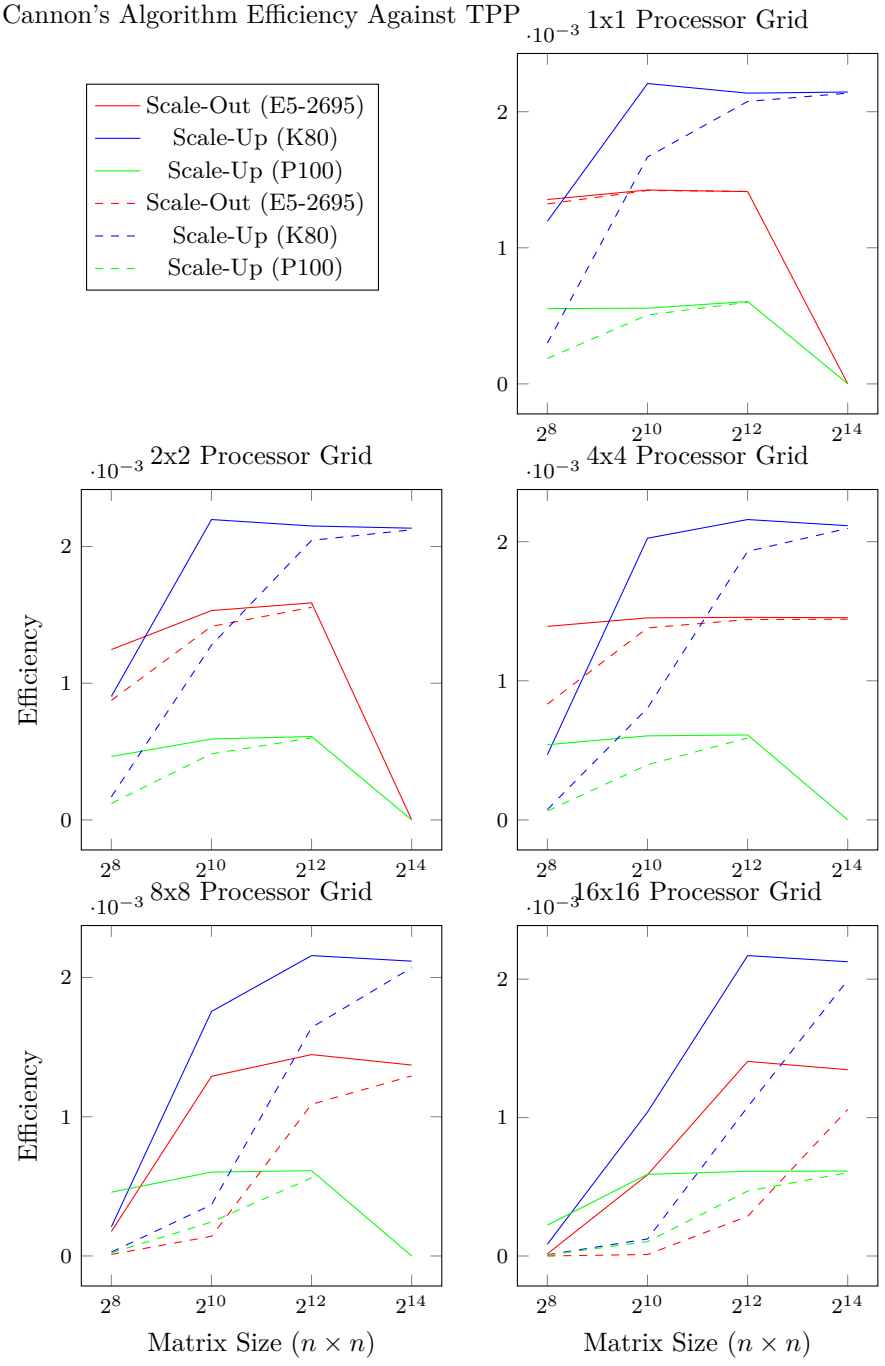
Efficiency (TPP) The efficiency of the scale-up and scale-out variants against theoretical peak performance (TPP) is calculated by the following formulas[16, p. 14]:

$$E_{TPP} = \frac{2mqn/t_{mult}}{P*R_{TPP}} \text{ (solid lines)}$$

$$E_{TPP} = \frac{2mqn/(t_{mult}+t_{comm})}{P*R_{TPP}} \text{ (dashed lines)}$$

With regard to TPP, the efficiency of both implementations of Cannon's algorithm is abysmal, barely reaching 0.2%, and only the K80 runs at that. The picture only worsens when one considers the communication time when calculating efficiency. It is interesting that the K80 would see the greatest efficiency, even over that of the P100. This could be explained by the fact that the current version of the PGI compiler targets Tesla through the `-ta=tesla,cuda-8.0` compiler flag. There is no similar flag for targeting Maxwell, Pascal, or, somewhat more understandably, Volta architectures. As a result, the code generated by the PGI compiler is most likely optimized for the K80 and cannot fully take advantage of the architectural innovations available in the hardware of the P100.

Graphs of the efficiency against TPP for the various cases are given below.

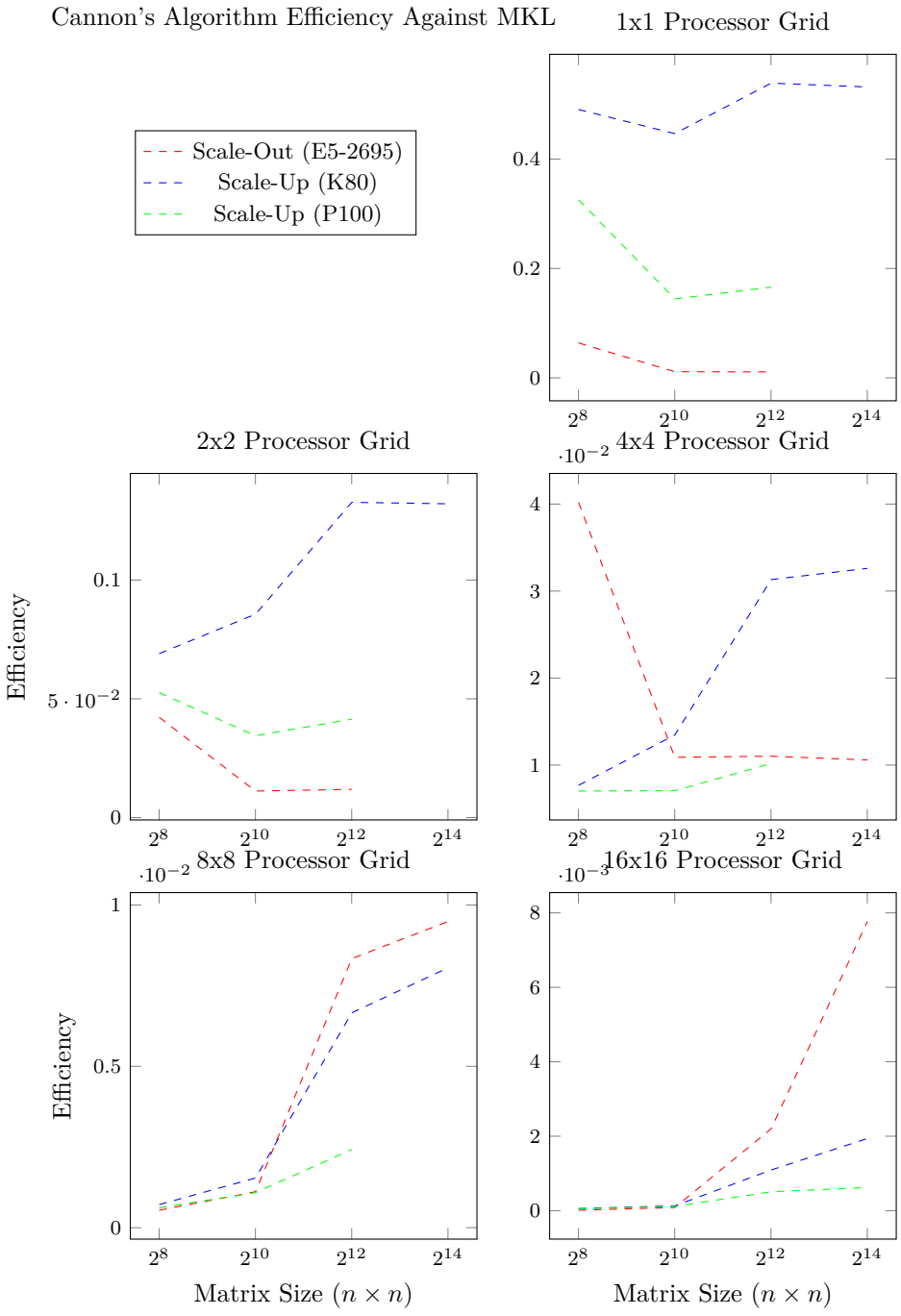

Cannon's Algorithm Efficiency Against TPP
Scale-Out (E5-2695)
Scale-Up (K80)
Scale-Up (P100)
Scale-Out (E5-2695)
Scale-Up (K80)
Scale-Up (P100)
1x1 Processor Grid
2x2 Processor Grid
4x4 Processor Grid
8x8 Processor Grid
16x16 Processor Grid
$\cdot 10^{-3}$
Efficiency
Matrix Size ($n \times n$)
0
1
2
2^8
2^{10}
2^{12}
2^{14}

Efficiency (MKL) Even though the implementations of Cannon's algorithm have shown so far to have underutilized the computing resources available, there are many reasons from operation imbalance to the inability to employ VLIW or SIMD tactics[16, pp. 4-5] as to why TPP is not practically achievable. It is often instructive to compare performance against the current state of the art to get a gauge on whether a strategy is worth pursuing or refining any further. It is clear from previous graphs that CUBLAS is far and away the optimal solution for the target matrix sizes discussed in this report, but it is worth examining whether the parallelization of Cannon's algorithm offers any benefit over the MKL single-core algorithm. The following formula is employed to provide a fair test of the true efficiency of the implementations[16, p. 14] under the presumption that the `mkl_cblas_sgemm` is the state of the art for single-core, single-precision matrix multiplication. Under a strong-scaling model, the best performance that can be reasonably expected is the rate of MKL's library function multiplied by the number of processors unless serendipities in the hardware provide a superlinear speedup.

$$E_{MKL} = \frac{2mqn/(t_{mult}+t_{comm})}{P*R_{MKL}} \text{ (dashed lines)}$$

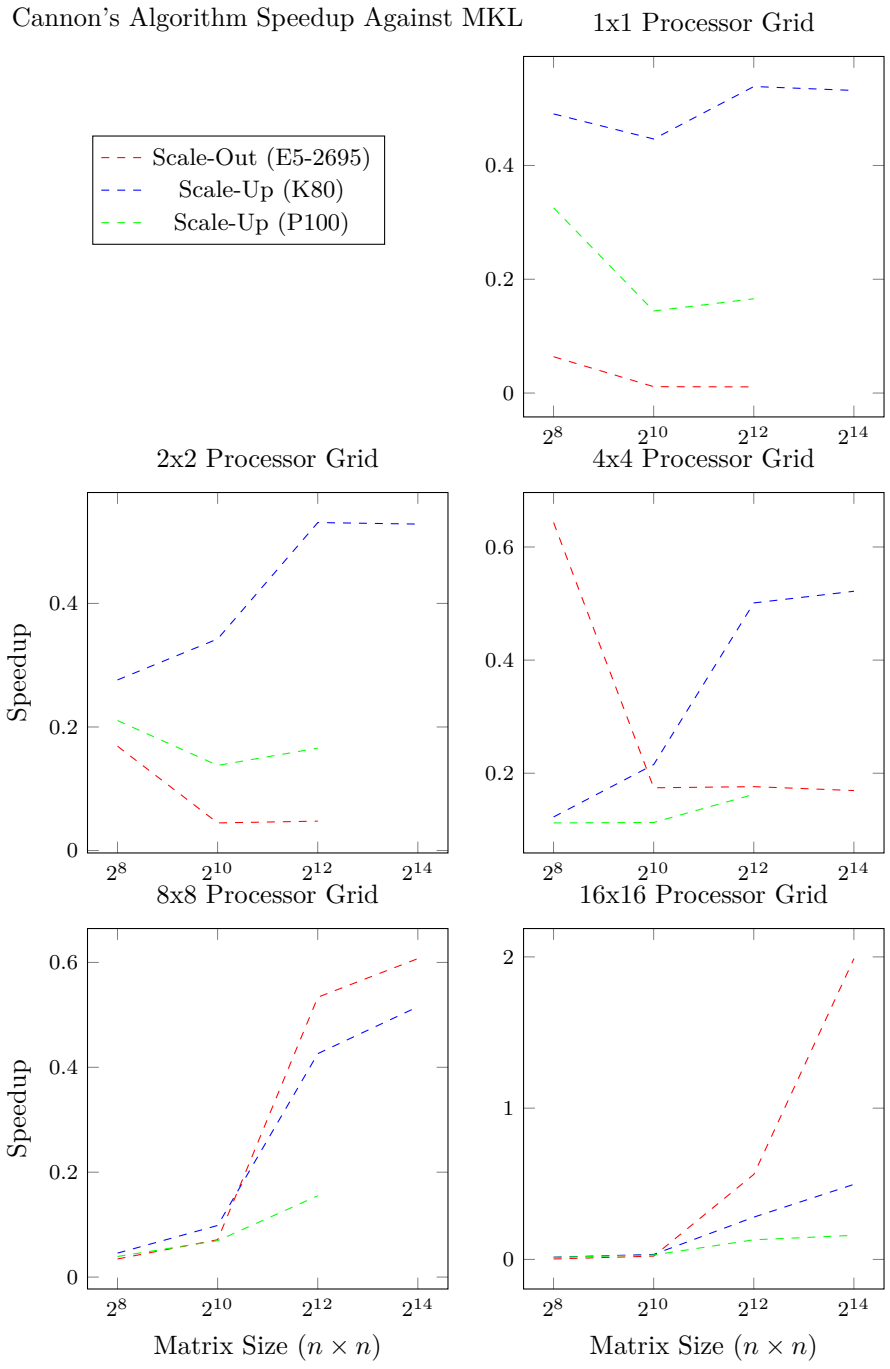
The equation involving only the multiplication time is not considered here since the communication overhead is an integral part of the time cost for Cannon's algorithm. It would be an unfair comparison considering MKL has no appreciable communication cost apart from that which may be incurred within the function call, and that cost is implicitly included in the measurements for the multiplication time reported by the `mkl_cblas_sgemm` harness.

Unfortunately, the efficiency calculated against a strong-scaling model using `mkl_cblas_sgemm` as the maximally performant single-core baseline is not much better than that calculated against TPP. The peaks occur with few processors on the grid at around 5-10%, but as more processors are added, it becomes clear that the resources are not being optimally used, and efficiency in the 256 processor 16x16 grid is at best around 0.8%.


Cannon's Algorithm Efficiency Against MKL
Scale-Out (E5-2695)
Scale-Up (K80)
Scale-Up (P100)
1x1 Processor Grid
2x2 Processor Grid
4x4 Processor Grid
8x8 Processor Grid
16x16 Processor Grid
Efficiency
Matrix Size ($n \times n$)
$\cdot 10^{-2}$
$\cdot 10^{-3}$
$5 \cdot 10^{-2}$
2^8
2^{10}
2^{12}
2^{14}

Speedup (MKL) Given the poor efficiency seen with regard to TPP and the single-core state of the art, it raises the question whether any speedup was achieved at all. Good practice dictates that parallel speedup be calculated with respect to the fraction of the program that undergoes parallelization. Since the timings measured only covered the actual matrix multiplication and the communication necessary to support that multiplication, the simple formula $S_{MKL} = \frac{t_{MKL}}{t_{mult}+t_{comm}}$ should suffice to give an appropriate speedup factor relative to the `mkl_cblas_sgemm` implementation. Again both the multiplication time and communication time are considered as integral and inextricable parts of the implementations of Cannon's algorithm in both the scale-up and scale-out variants.

As the graphs below demonstrate, the speedup results are nearly as unfavorable as the efficiency results. In all cases save one, the MKL version outperformed all variants of the Cannon's algorithm implementations. The one counterexample was the 16384x16384 matrix case, and it required throwing 256 cores to cut the time in half that the MKL version took to achieve with one core. The expenditure in terms of resources does not justify the results.


Cannon's Algorithm Speedup Against MKL
Scale-Out (E5-2695)
Scale-Up (K80)
Scale-Up (P100)
1x1 Processor Grid
2x2 Processor Grid
4x4 Processor Grid
8x8 Processor Grid
16x16 Processor Grid
Speedup
Matrix Size ($n \times n$)
0
0.2
0.4
0.6
1
2
2^8
2^{10}
2^{12}
2^{14}

6 Conclusions

Based on the experimental process that was undertaken and the results gathered therefrom, it appears that Cannon's algorithm as implemented was a poor choice for this exercise owing to the inflexibility of the algorithm as originally defined. A major part of this shortcoming had its roots to the author's relative inexperience with OpenACC and MPI. There are likely better means of structuring the communications so as not to occupy so much time and better ways to write the GPU directives.

It would also have behooved the author to try and implement the generalized version of Cannon's algorithm covered in lecture [17] rather than the more rigid block-oriented version as presented by Gupta and Sadayappan[15]. The former would likely have offered more opportunities for pipelining and enabled the author to pursue the stated goal of covering the effect of the algorithm on matrices with extreme rectangular skew.

That being said, the facilities within OpenACC and OpenMP leave quite a bit to be desired in terms of fine control over offloading to accelerators. The last segment of the experimental process became more of a game of outwitting the compiler than setting the processor grid up appropriately on the accelerator. In retrospect, CUDA would have probably been the better choice for implementing something so structure-oriented as Cannon's algorithm on the GPU. It would have limited the program to running on NVIDIA GPUs, but that constraint was already in place from the moment that `-ta=tesla` was put in the Makefile along with the NVIDIA specific device type directives and initializers in the C source.

As for the results received, it seems that Cannon's algorithm is better suited to the scale-out case in general. The algorithm more naturally maps onto a MPI distributed memory framework rather than a shared memory model like OpenMP or OpenACC. One might be inclined to conclude that pairing a distributed MPI network of GPU-accelerated nodes would give an implementation of the algorithm the necessary shot-in-the-arm, but the frequency of communication would be exacerbated by a two-stage extraction from device to host and then host to network followed by the laborious translation from network to target host and then to target GPU. Unless the GPU card was directly connected to other GPUs in the processor network and transfers could be made on something faster than a PCI bus, the purported benefits would soon evaporate.

Direction for future work could include a refactoring to either use the generalized form of Cannon's algorithm or perhaps another algorithm altogether such as the Scalable Universal Matrix Multiplication Algorithm (SUMMA)[4] which is apparently used in several linear algebra libraries. Furthermore, a better

understanding of MPI communication and GPU processing contexts and minimizing the concomitant overhead in both cases would prove valuable in future efforts, regardless of the algorithm chosen.

Unfortunately, the facilities for taking power measurements were not fully available given the permissions restrictions on BRIDGES with respect to the RAPL component of PAPI. Some thought was given to employing the `rapl_read` executable wrapper in lieu of PAPI RAPL measurements at the function level, but the strategy was rejected for lack of time and the author's uncertainty as to how such a wrapper would behave with regard to the MPI scale-out case. A better solution in the form of a power-measurement library that could work within the context of production cluster environments would provide some much-needed insight to this critical yet often overlooked component of software development. This would be a fascinating research project in its own right and one sorely needed in light of the author's own experience.

7 Acknowledgments

The author is indebted to the LaTeX project[20] and the following LaTeX package and tutorial authors:

- Dr. Christian Feuersänger for his excellent pgfplots[12] and pgfplotstable[13] packages
- Philip Lehman et al. for their contributions to the biblatex[24] package which provides a number of improvements over the venerable but outdated bibtex[11, 10] package
- Philip Kime and François Charette for building the biber [19] backend for biblatex
- Geoffrey M. Poore for the syntax highlighting made possible through his minted[28] package in conjunction with Pygments
- David Carlisle for the fundamental enumerate[5] package
- Johannes Braams for the useful and flexible alltt[2] package
- the anonymous contributors from the Wikimedia Project for maintaining the LaTeX Wikibooks collection of tutorials[23, 22, 21, 1]
- Nadir Soualem and M. Tibbits for their helpful posts on getting Beamer to generate presentation slides[31, 34]
- John Kormylo for his helpful response to a post on creating a common legend for pgfplots graphs[35]

The author is also indebted to Dr. Lennart Johnsson, Dr. Amit Amritkar, and Dr. Edgar Gabriel for their contribution to the lectures and instruction for the Introduction to High Performance Computing course. Finally, the author would like to thank his wife and daughter, as well as his employer, SixFoot LLC, for their patience, understanding, and support of him in the pursuit of furthering his education.

8 Appendix A: Scale-Up Raw Results (K80)

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
1	256	256	256	10	0	$3.36 \cdot 10^7$	10.57	3.17	33.54
1	256	256	256	10	1	$3.36 \cdot 10^7$	9.44	3.55	6.82
1	256	256	256	10	2	$3.36 \cdot 10^7$	11.93	2.81	7.15
1	256	256	256	10	3	$3.36 \cdot 10^7$	9.89	3.39	6.95
1	256	256	256	10	4	$3.36 \cdot 10^7$	10.94	3.07	6.85
1	256	256	256	10	5	$3.36 \cdot 10^7$	9.74	3.45	6.86
1	256	256	256	10	6	$3.36 \cdot 10^7$	10.85	3.09	6.9
1	256	256	256	10	7	$3.36 \cdot 10^7$	9.83	3.41	6.86
1	256	256	256	10	8	$3.36 \cdot 10^7$	10.8	3.11	6.91
1	256	256	256	10	9	$3.36 \cdot 10^7$	10.43	3.22	7.06
1	1,024	1,024	1,024	10	0	$2.15 \cdot 10^9$	19.2	111.86	64.1
1	1,024	1,024	1,024	10	1	$2.15 \cdot 10^9$	19.33	111.07	32.73
1	1,024	1,024	1,024	10	2	$2.15 \cdot 10^9$	19.38	110.79	32.65
1	1,024	1,024	1,024	10	3	$2.15 \cdot 10^9$	19.35	110.95	32.3
1	1,024	1,024	1,024	10	4	$2.15 \cdot 10^9$	19.3	111.25	32.41
1	1,024	1,024	1,024	10	5	$2.15 \cdot 10^9$	19.32	111.13	32.35
1	1,024	1,024	1,024	10	6	$2.15 \cdot 10^9$	19.27	111.47	35.03
1	1,024	1,024	1,024	10	7	$2.15 \cdot 10^9$	19.25	111.56	32.86
1	1,024	1,024	1,024	10	8	$2.15 \cdot 10^9$	19.24	111.62	33.1
1	1,024	1,024	1,024	10	9	$2.15 \cdot 10^9$	19.28	111.36	33.07
1	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	18.67	7,360.6	240.81
1	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	18.67	7,361.38	209.75
1	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	18.66	7,363.99	209.53
1	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	18.7	7,347.92	209.6
1	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	18.66	7,364.05	213.6
1	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	18.68	7,358.75	210.71
1	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	18.67	7,360.34	231.51
1	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	18.64	7,372.71	209.61
1	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	18.68	7,358.72	209.92
1	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	18.69	7,354.92	209.37
1	16,384	16,384	16,384	10	0	$8.8 \cdot 10^{12}$	18.74	$4.69 \cdot 10^5$	1,883.04
1	16,384	16,384	16,384	10	1	$8.8 \cdot 10^{12}$	18.74	$4.69 \cdot 10^5$	1,840.11
1	16,384	16,384	16,384	10	2	$8.8 \cdot 10^{12}$	18.75	$4.69 \cdot 10^5$	1,835.25
1	16,384	16,384	16,384	10	3	$8.8 \cdot 10^{12}$	18.76	$4.69 \cdot 10^5$	1,836.13

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
1	16,384	16,384	16,384	10	4	$8.8 \cdot 10^{12}$	18.75	$4.69 \cdot 10^{5}$	1,837.79
1	16,384	16,384	16,384	10	5	$8.8 \cdot 10^{12}$	18.75	$4.69 \cdot 10^{5}$	1,840.23
1	16,384	16,384	16,384	10	6	$8.8 \cdot 10^{12}$	18.75	$4.69 \cdot 10^{5}$	1,839.89
1	16,384	16,384	16,384	10	7	$8.8 \cdot 10^{12}$	18.73	$4.7 \cdot 10^{5}$	1,842.99
1	16,384	16,384	16,384	10	8	$8.8 \cdot 10^{12}$	18.74	$4.69 \cdot 10^{5}$	1,833.94
1	16,384	16,384	16,384	10	9	$8.8 \cdot 10^{12}$	18.74	$4.69 \cdot 10^{5}$	1,838.51
4	256	256	256	10	0	$3.36 \cdot 10^{7}$	6.86	4.89	45.84
4	256	256	256	10	1	$3.36 \cdot 10^{7}$	7.91	4.24	15.73
4	256	256	256	10	2	$3.36 \cdot 10^{7}$	7.59	4.42	15.34
4	256	256	256	10	3	$3.36 \cdot 10^{7}$	7.54	4.45	15.29
4	256	256	256	10	4	$3.36 \cdot 10^{7}$	7.63	4.4	15.27
4	256	256	256	10	5	$3.36 \cdot 10^{7}$	7.86	4.27	15.31
4	256	256	256	10	6	$3.36 \cdot 10^{7}$	8.13	4.13	15.34
4	256	256	256	10	7	$3.36 \cdot 10^{7}$	8.2	4.09	15.82
4	256	256	256	10	8	$3.36 \cdot 10^{7}$	8.42	3.99	15.63
4	256	256	256	10	9	$3.36 \cdot 10^{7}$	8.84	3.8	15.45
4	1,024	1,024	1,024	10	0	$2.15 \cdot 10^{9}$	19.07	112.6	112.91
4	1,024	1,024	1,024	10	1	$2.15 \cdot 10^{9}$	19.17	112.04	76.96
4	1,024	1,024	1,024	10	2	$2.15 \cdot 10^{9}$	19.38	110.78	76.92
4	1,024	1,024	1,024	10	3	$2.15 \cdot 10^{9}$	19.14	112.17	77.67
4	1,024	1,024	1,024	10	4	$2.15 \cdot 10^{9}$	19.12	112.3	75.36
4	1,024	1,024	1,024	10	5	$2.15 \cdot 10^{9}$	19.19	111.93	75.66
4	1,024	1,024	1,024	10	6	$2.15 \cdot 10^{9}$	19.22	111.74	75.67
4	1,024	1,024	1,024	10	7	$2.15 \cdot 10^{9}$	19.16	112.1	76.02
4	1,024	1,024	1,024	10	8	$2.15 \cdot 10^{9}$	19.32	111.15	79.78
4	1,024	1,024	1,024	10	9	$2.15 \cdot 10^{9}$	19.27	111.44	76.08
4	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	18.81	7,305.84	401.07
4	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	18.79	7,315.38	371.47
4	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	18.8	7,309.78	371.86
4	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	18.8	7,309	373.68
4	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	18.79	7,316.36	371.14
4	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	18.78	7,318.39	371.61
4	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	18.79	7,312.9	374.01
4	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	18.78	7,318.45	371.04
4	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	18.79	7,315.74	370.72
4	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	18.8	7,311.16	375.09
4	16,384	16,384	16,384	10	0	$8.8 \cdot 10^{12}$	18.71	$4.7 \cdot 10^{5}$	2,610.2
4	16,384	16,384	16,384	10	1	$8.8 \cdot 10^{12}$	18.66	$4.71 \cdot 10^{5}$	2,595.26
4	16,384	16,384	16,384	10	2	$8.8 \cdot 10^{12}$	18.67	$4.71 \cdot 10^{5}$	2,577.96
4	16,384	16,384	16,384	10	3	$8.8 \cdot 10^{12}$	18.66	$4.71 \cdot 10^{5}$	2,565.4
4	16,384	16,384	16,384	10	4	$8.8 \cdot 10^{12}$	18.65	$4.72 \cdot 10^{5}$	2,568.04
4	16,384	16,384	16,384	10	5	$8.8 \cdot 10^{12}$	18.68	$4.71 \cdot 10^{5}$	2,562.55
4	16,384	16,384	16,384	10	6	$8.8 \cdot 10^{12}$	18.62	$4.72 \cdot 10^{5}$	2,563.3
4	16,384	16,384	16,384	10	7	$8.8 \cdot 10^{12}$	18.62	$4.72 \cdot 10^{5}$	2,563.19
4	16,384	16,384	16,384	10	8	$8.8 \cdot 10^{12}$	18.61	$4.73 \cdot 10^{5}$	2,561.96

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
4	16,384	16,384	16,384	10	9	$8.8 \cdot 10^{12}$	18.65	$4.72 \cdot 10^{5}$	2,566.99
16	256	256	256	10	0	$3.36 \cdot 10^{7}$	3.53	9.5	70.88
16	256	256	256	10	1	$3.36 \cdot 10^{7}$	4.08	8.22	39.45
16	256	256	256	10	2	$3.36 \cdot 10^{7}$	4.2	7.99	39.26
16	256	256	256	10	3	$3.36 \cdot 10^{7}$	4.03	8.34	42.85
16	256	256	256	10	4	$3.36 \cdot 10^{7}$	4.3	7.8	39.25
16	256	256	256	10	5	$3.36 \cdot 10^{7}$	4.17	8.05	39.61
16	256	256	256	10	6	$3.36 \cdot 10^{7}$	4.04	8.31	40.02
16	256	256	256	10	7	$3.36 \cdot 10^{7}$	4.38	7.65	39.25
16	256	256	256	10	8	$3.36 \cdot 10^{7}$	4.21	7.98	39.8
16	256	256	256	10	9	$3.36 \cdot 10^{7}$	4.07	8.25	39.88
16	1,024	1,024	1,024	10	0	$2.15 \cdot 10^{9}$	17.55	122.35	211.72
16	1,024	1,024	1,024	10	1	$2.15 \cdot 10^{9}$	17.63	121.83	180.07
16	1,024	1,024	1,024	10	2	$2.15 \cdot 10^{9}$	17.76	120.92	182.31
16	1,024	1,024	1,024	10	3	$2.15 \cdot 10^{9}$	17.69	121.43	179.89
16	1,024	1,024	1,024	10	4	$2.15 \cdot 10^{9}$	17.78	120.8	180.75
16	1,024	1,024	1,024	10	5	$2.15 \cdot 10^{9}$	17.78	120.81	181
16	1,024	1,024	1,024	10	6	$2.15 \cdot 10^{9}$	17.68	121.5	180.56
16	1,024	1,024	1,024	10	7	$2.15 \cdot 10^{9}$	17.68	121.47	180.76
16	1,024	1,024	1,024	10	8	$2.15 \cdot 10^{9}$	17.7	121.3	180.74
16	1,024	1,024	1,024	10	9	$2.15 \cdot 10^{9}$	17.72	121.16	183.92
16	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	18.88	7,278.92	882.51
16	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	18.88	7,281.03	850.4
16	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	18.87	7,281.66	877.66
16	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	18.87	7,285.07	890.93
16	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	18.86	7,285.6	856.69
16	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	18.87	7,284.43	852.88
16	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	18.87	7,285.17	850.48
16	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	18.87	7,283.72	851.11
16	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	18.87	7,284.71	851.07
16	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	18.87	7,283.58	854.4
16	16,384	16,384	16,384	10	0	$8.8 \cdot 10^{12}$	18.6	$4.73 \cdot 10^{5}$	4,353.9
16	16,384	16,384	16,384	10	1	$8.8 \cdot 10^{12}$	18.53	$4.75 \cdot 10^{5}$	4,334.34
16	16,384	16,384	16,384	10	2	$8.8 \cdot 10^{12}$	18.47	$4.76 \cdot 10^{5}$	4,334.95
16	16,384	16,384	16,384	10	3	$8.8 \cdot 10^{12}$	18.45	$4.77 \cdot 10^{5}$	4,335.96
16	16,384	16,384	16,384	10	4	$8.8 \cdot 10^{12}$	18.45	$4.77 \cdot 10^{5}$	4,331.7
16	16,384	16,384	16,384	10	5	$8.8 \cdot 10^{12}$	18.46	$4.76 \cdot 10^{5}$	4,335.5
16	16,384	16,384	16,384	10	6	$8.8 \cdot 10^{12}$	18.47	$4.76 \cdot 10^{5}$	4,329.42
16	16,384	16,384	16,384	10	7	$8.8 \cdot 10^{12}$	18.47	$4.76 \cdot 10^{5}$	4,323.87
16	16,384	16,384	16,384	10	8	$8.8 \cdot 10^{12}$	18.47	$4.76 \cdot 10^{5}$	4,329.26
16	16,384	16,384	16,384	10	9	$8.8 \cdot 10^{12}$	18.47	$4.76 \cdot 10^{5}$	4,338.22
64	256	256	256	10	0	$3.36 \cdot 10^{7}$	1.77	18.91	148.44
64	256	256	256	10	1	$3.36 \cdot 10^{7}$	1.82	18.45	115.8
64	256	256	256	10	2	$3.36 \cdot 10^{7}$	1.88	17.8	119.66
64	256	256	256	10	3	$3.36 \cdot 10^{7}$	1.83	18.3	115.58

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
64	256	256	256	10	4	$3.36 \cdot 10^7$	1.87	17.98	116.59
64	256	256	256	10	5	$3.36 \cdot 10^7$	1.85	18.12	115.23
64	256	256	256	10	6	$3.36 \cdot 10^7$	1.81	18.54	115.33
64	256	256	256	10	7	$3.36 \cdot 10^7$	1.81	18.55	114.75
64	256	256	256	10	8	$3.36 \cdot 10^7$	1.8	18.6	115.42
64	256	256	256	10	9	$3.36 \cdot 10^7$	1.85	18.17	115.86
64	1,024	1,024	1,024	10	0	$2.15 \cdot 10^9$	15.32	140.2	557.59
64	1,024	1,024	1,024	10	1	$2.15 \cdot 10^9$	15.34	140.03	525.7
64	1,024	1,024	1,024	10	2	$2.15 \cdot 10^9$	15.29	140.46	524.25
64	1,024	1,024	1,024	10	3	$2.15 \cdot 10^9$	15.6	137.64	527.87
64	1,024	1,024	1,024	10	4	$2.15 \cdot 10^9$	15.39	139.53	525.84
64	1,024	1,024	1,024	10	5	$2.15 \cdot 10^9$	15.26	140.75	523.92
64	1,024	1,024	1,024	10	6	$2.15 \cdot 10^9$	15.41	139.35	525.76
64	1,024	1,024	1,024	10	7	$2.15 \cdot 10^9$	15.31	140.29	525.76
64	1,024	1,024	1,024	10	8	$2.15 \cdot 10^9$	15.2	141.25	523.79
64	1,024	1,024	1,024	10	9	$2.15 \cdot 10^9$	15.37	139.71	525.61
64	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	18.88	7,278.75	2,318.21
64	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	18.87	7,285.26	2,279.12
64	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	18.86	7,285.46	2,289.96
64	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	18.87	7,284.89	2,295.55
64	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	18.85	7,290.43	2,318.95
64	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	18.85	7,290.35	2,283.37
64	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	18.85	7,290.64	2,281.74
64	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	18.84	7,293.26	2,285.36
64	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	18.85	7,291.5	2,281.56
64	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	18.85	7,289.87	2,288.45
64	16,384	16,384	16,384	10	0	$8.8 \cdot 10^{12}$	18.49	$4.76 \cdot 10^5$	10,302.69
64	16,384	16,384	16,384	10	1	$8.8 \cdot 10^{12}$	18.5	$4.76 \cdot 10^5$	10,417.99
64	16,384	16,384	16,384	10	2	$8.8 \cdot 10^{12}$	18.5	$4.75 \cdot 10^5$	11,032.66
64	16,384	16,384	16,384	10	3	$8.8 \cdot 10^{12}$	18.51	$4.75 \cdot 10^5$	11,436.06
64	16,384	16,384	16,384	10	4	$8.8 \cdot 10^{12}$	18.52	$4.75 \cdot 10^5$	10,790.82
64	16,384	16,384	16,384	10	5	$8.8 \cdot 10^{12}$	18.5	$4.75 \cdot 10^5$	10,752.42
64	16,384	16,384	16,384	10	6	$8.8 \cdot 10^{12}$	18.5	$4.76 \cdot 10^5$	10,751.17
64	16,384	16,384	16,384	10	7	$8.8 \cdot 10^{12}$	18.52	$4.75 \cdot 10^5$	10,731.04
64	16,384	16,384	16,384	10	8	$8.8 \cdot 10^{12}$	18.52	$4.75 \cdot 10^5$	10,755.64
64	16,384	16,384	16,384	10	9	$8.8 \cdot 10^{12}$	18.52	$4.75 \cdot 10^5$	10,734.75
256	256	256	256	10	0	$3.36 \cdot 10^7$	0.76	44.18	415.88
256	256	256	256	10	1	$3.36 \cdot 10^7$	0.74	45.55	379.92
256	256	256	256	10	2	$3.36 \cdot 10^7$	0.72	46.77	378.62
256	256	256	256	10	3	$3.36 \cdot 10^7$	0.74	45.3	381.32
256	256	256	256	10	4	$3.36 \cdot 10^7$	0.74	45.51	379.37
256	256	256	256	10	5	$3.36 \cdot 10^7$	0.77	43.84	381.15
256	256	256	256	10	6	$3.36 \cdot 10^7$	0.73	46.04	379.01
256	256	256	256	10	7	$3.36 \cdot 10^7$	0.76	44.29	381
256	256	256	256	10	8	$3.36 \cdot 10^7$	0.74	45.15	379.24

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
256	256	256	256	10	9	$3.36 \cdot 10^7$	0.75	44.98	381.54
256	1,024	1,024	1,024	10	0	$2.15 \cdot 10^9$	9.11	235.77	1,789.96
256	1,024	1,024	1,024	10	1	$2.15 \cdot 10^9$	9.15	234.63	1,757.23
256	1,024	1,024	1,024	10	2	$2.15 \cdot 10^9$	9.19	233.73	1,758.54
256	1,024	1,024	1,024	10	3	$2.15 \cdot 10^9$	9.07	236.66	1,757.4
256	1,024	1,024	1,024	10	4	$2.15 \cdot 10^9$	9.06	236.91	1,758.45
256	1,024	1,024	1,024	10	5	$2.15 \cdot 10^9$	9.08	236.57	1,758.81
256	1,024	1,024	1,024	10	6	$2.15 \cdot 10^9$	9.05	237.29	1,755.46
256	1,024	1,024	1,024	10	7	$2.15 \cdot 10^9$	9.03	237.86	1,755.76
256	1,024	1,024	1,024	10	8	$2.15 \cdot 10^9$	9.05	237.3	1,757.15
256	1,024	1,024	1,024	10	9	$2.15 \cdot 10^9$	9.09	236.2	1,756.29
256	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	19.29	7,124.37	7,385.47
256	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	19.2	7,157.87	7,347.69
256	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	19.13	7,184.15	7,350.33
256	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	19.01	7,228.26	7,351.29
256	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	18.99	7,237.38	7,347.01
256	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	18.98	7,242.49	7,355.91
256	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	18.88	7,280.27	7,351.56
256	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	18.75	7,328.16	7,351.18
256	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	18.73	7,337.01	7,353.51
256	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	18.72	7,339.96	7,358.99
256	16,384	16,384	16,384	10	0	$8.8 \cdot 10^{12}$	18.57	$4.74 \cdot 10^5$	31,905.28
256	16,384	16,384	16,384	10	1	$8.8 \cdot 10^{12}$	18.56	$4.74 \cdot 10^5$	31,653.06
256	16,384	16,384	16,384	10	2	$8.8 \cdot 10^{12}$	18.58	$4.73 \cdot 10^5$	30,779.39
256	16,384	16,384	16,384	10	3	$8.8 \cdot 10^{12}$	18.58	$4.73 \cdot 10^5$	30,743.19
256	16,384	16,384	16,384	10	4	$8.8 \cdot 10^{12}$	18.58	$4.73 \cdot 10^5$	31,248.28
256	16,384	16,384	16,384	10	5	$8.8 \cdot 10^{12}$	18.58	$4.73 \cdot 10^5$	31,721.8
256	16,384	16,384	16,384	10	6	$8.8 \cdot 10^{12}$	18.58	$4.73 \cdot 10^5$	31,175.16
256	16,384	16,384	16,384	10	7	$8.8 \cdot 10^{12}$	18.59	$4.73 \cdot 10^5$	31,195.21
256	16,384	16,384	16,384	10	8	$8.8 \cdot 10^{12}$	18.59	$4.73 \cdot 10^5$	31,765.05
256	16,384	16,384	16,384	10	9	$8.8 \cdot 10^{12}$	18.58	$4.74 \cdot 10^5$	31,754.37

9 Appendix B: Scale-Up Raw Results (P100)

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
1	256	256	256	10	0	$3.36 \cdot 10^7$	5.13	6.55	48.05
1	256	256	256	10	1	$3.36 \cdot 10^7$	5.15	6.51	10.43
1	256	256	256	10	2	$3.36 \cdot 10^7$	5.14	6.53	8.91
1	256	256	256	10	3	$3.36 \cdot 10^7$	5.15	6.52	10.9
1	256	256	256	10	4	$3.36 \cdot 10^7$	5.15	6.52	8.52
1	256	256	256	10	5	$3.36 \cdot 10^7$	5.15	6.51	8.16
1	256	256	256	10	6	$3.36 \cdot 10^7$	5.15	6.52	8.35
1	256	256	256	10	7	$3.36 \cdot 10^7$	5.15	6.52	8.34
1	256	256	256	10	8	$3.36 \cdot 10^7$	5.15	6.52	8.2

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
1	256	256	256	10	9	$3.36 \cdot 10^7$	5.14	6.52	8.1
1	1,024	1,024	1,024	10	0	$2.15 \cdot 10^9$	5.17	415.07	73.53
1	1,024	1,024	1,024	10	1	$2.15 \cdot 10^9$	5.17	415.16	34.18
1	1,024	1,024	1,024	10	2	$2.15 \cdot 10^9$	5.17	414.97	30.25
1	1,024	1,024	1,024	10	3	$2.15 \cdot 10^9$	5.17	415.15	43.41
1	1,024	1,024	1,024	10	4	$2.15 \cdot 10^9$	5.17	415.1	40.88
1	1,024	1,024	1,024	10	5	$2.15 \cdot 10^9$	5.18	414.97	35.7
1	1,024	1,024	1,024	10	6	$2.15 \cdot 10^9$	5.17	414.99	38.73
1	1,024	1,024	1,024	10	7	$2.15 \cdot 10^9$	5.17	415.03	33.08
1	1,024	1,024	1,024	10	8	$2.15 \cdot 10^9$	5.18	414.9	46.25
1	1,024	1,024	1,024	10	9	$2.15 \cdot 10^9$	5.17	415.13	34.47
1	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	5.63	24,414.76	272.37
1	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	5.63	24,411.05	215.7
1	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	5.63	24,411.19	188.37
1	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	5.63	24,400.85	242.69
1	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	5.63	24,400.18	220.42
1	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	5.63	24,403.14	212.25
1	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	5.63	24,400.32	202.59
1	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	5.63	24,412.48	217.46
1	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	5.63	24,417.16	225.25
1	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	5.63	24,401.43	231.67
4	256	256	256	10	0	$3.36 \cdot 10^7$	4.27	7.85	56.79
4	256	256	256	10	1	$3.36 \cdot 10^7$	4.12	8.14	17.22
4	256	256	256	10	2	$3.36 \cdot 10^7$	4.29	7.82	17.04
4	256	256	256	10	3	$3.36 \cdot 10^7$	4.44	7.56	17.84
4	256	256	256	10	4	$3.36 \cdot 10^7$	4.29	7.82	17.67
4	256	256	256	10	5	$3.36 \cdot 10^7$	4.29	7.82	17.46
4	256	256	256	10	6	$3.36 \cdot 10^7$	4.44	7.56	17.12
4	256	256	256	10	7	$3.36 \cdot 10^7$	4.44	7.56	20.4
4	256	256	256	10	8	$3.36 \cdot 10^7$	4.29	7.82	17
4	256	256	256	10	9	$3.36 \cdot 10^7$	4.28	7.84	22.48
4	1,024	1,024	1,024	10	0	$2.15 \cdot 10^9$	5.51	389.61	142.7
4	1,024	1,024	1,024	10	1	$2.15 \cdot 10^9$	5.55	386.59	84.04
4	1,024	1,024	1,024	10	2	$2.15 \cdot 10^9$	5.47	392.45	89.27
4	1,024	1,024	1,024	10	3	$2.15 \cdot 10^9$	5.51	389.58	73.65
4	1,024	1,024	1,024	10	4	$2.15 \cdot 10^9$	5.51	389.63	88.27
4	1,024	1,024	1,024	10	5	$2.15 \cdot 10^9$	5.51	389.45	95.93
4	1,024	1,024	1,024	10	6	$2.15 \cdot 10^9$	5.51	389.46	78.57
4	1,024	1,024	1,024	10	7	$2.15 \cdot 10^9$	5.51	389.55	83.67
4	1,024	1,024	1,024	10	8	$2.15 \cdot 10^9$	5.47	392.3	70.4
4	1,024	1,024	1,024	10	9	$2.15 \cdot 10^9$	5.51	389.64	70.44
4	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	5.67	24,222.14	446.45
4	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	5.67	24,221.05	393.04
4	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	5.68	24,214.11	408.69
4	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	5.68	24,216.42	404.82

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
4	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	5.68	24,212.9	374.22
4	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	5.68	24,214.88	375.66
4	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	5.66	24,295.18	416.99
4	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	5.67	24,218.81	362.56
4	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	5.66	24,293.93	377.58
4	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	5.68	24,215.1	375.39
16	256	256	256	10	0	$3.36 \cdot 10^{7}$	4.78	7.02	96.17
16	256	256	256	10	1	$3.36 \cdot 10^{7}$	4.76	7.05	54.38
16	256	256	256	10	2	$3.36 \cdot 10^{7}$	5.13	6.54	51.14
16	256	256	256	10	3	$3.36 \cdot 10^{7}$	5.13	6.54	55.79
16	256	256	256	10	4	$3.36 \cdot 10^{7}$	5.01	6.69	45.94
16	256	256	256	10	5	$3.36 \cdot 10^{7}$	5.09	6.6	36.22
16	256	256	256	10	6	$3.36 \cdot 10^{7}$	5.15	6.51	35.47
16	256	256	256	10	7	$3.36 \cdot 10^{7}$	5.15	6.52	42.55
16	256	256	256	10	8	$3.36 \cdot 10^{7}$	4.95	6.77	38.31
16	256	256	256	10	9	$3.36 \cdot 10^{7}$	5.13	6.54	37.89
16	1,024	1,024	1,024	10	0	$2.15 \cdot 10^{9}$	5.62	381.94	249.64
16	1,024	1,024	1,024	10	1	$2.15 \cdot 10^{9}$	5.62	381.86	209.7
16	1,024	1,024	1,024	10	2	$2.15 \cdot 10^{9}$	5.62	381.84	202.4
16	1,024	1,024	1,024	10	3	$2.15 \cdot 10^{9}$	5.62	381.89	184.44
16	1,024	1,024	1,024	10	4	$2.15 \cdot 10^{9}$	5.62	381.88	194.73
16	1,024	1,024	1,024	10	5	$2.15 \cdot 10^{9}$	5.62	381.88	196.6
16	1,024	1,024	1,024	10	6	$2.15 \cdot 10^{9}$	5.62	381.92	190.11
16	1,024	1,024	1,024	10	7	$2.15 \cdot 10^{9}$	5.62	381.88	182.36
16	1,024	1,024	1,024	10	8	$2.15 \cdot 10^{9}$	5.62	381.87	203.84
16	1,024	1,024	1,024	10	9	$2.15 \cdot 10^{9}$	5.62	381.86	192.44
16	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	5.68	24,204.4	1,001.76
16	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	5.68	24,204.98	942.75
16	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	5.68	24,202.62	871.81
16	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	5.68	24,204.94	897.95
16	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	5.68	24,184.23	849.6
16	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	5.68	24,203.96	816.99
16	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	5.68	24,185.59	898.19
16	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	5.68	24,204.67	825.53
16	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	5.68	24,185.89	796.22
16	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	5.68	24,184.16	898.71
64	256	256	256	10	0	$3.36 \cdot 10^{7}$	4.27	7.85	191.44
64	256	256	256	10	1	$3.36 \cdot 10^{7}$	4.25	7.9	152.03
64	256	256	256	10	2	$3.36 \cdot 10^{7}$	4.3	7.81	145.23
64	256	256	256	10	3	$3.36 \cdot 10^{7}$	4.23	7.94	156.15
64	256	256	256	10	4	$3.36 \cdot 10^{7}$	4.29	7.82	146.6
64	256	256	256	10	5	$3.36 \cdot 10^{7}$	4.24	7.91	155.6
64	256	256	256	10	6	$3.36 \cdot 10^{7}$	4.22	7.95	150.42
64	256	256	256	10	7	$3.36 \cdot 10^{7}$	4.26	7.87	146.95
64	256	256	256	10	8	$3.36 \cdot 10^{7}$	4.26	7.88	129.43

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
64	256	256	256	10	9	$3.36 \cdot 10^7$	4.26	7.88	146.59
64	1,024	1,024	1,024	10	0	$2.15 \cdot 10^9$	5.6	383.14	663.5
64	1,024	1,024	1,024	10	1	$2.15 \cdot 10^9$	5.6	383.68	582.8
64	1,024	1,024	1,024	10	2	$2.15 \cdot 10^9$	5.61	383.11	575.78
64	1,024	1,024	1,024	10	3	$2.15 \cdot 10^9$	5.61	383.06	550.21
64	1,024	1,024	1,024	10	4	$2.15 \cdot 10^9$	5.6	383.21	517.67
64	1,024	1,024	1,024	10	5	$2.15 \cdot 10^9$	5.6	383.38	538.9
64	1,024	1,024	1,024	10	6	$2.15 \cdot 10^9$	5.61	382.82	548.13
64	1,024	1,024	1,024	10	7	$2.15 \cdot 10^9$	5.61	382.51	571.96
64	1,024	1,024	1,024	10	8	$2.15 \cdot 10^9$	5.61	383.08	560.09
64	1,024	1,024	1,024	10	9	$2.15 \cdot 10^9$	5.6	383.69	534.91
64	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	5.69	24,133.61	2,263.17
64	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	5.7	24,129.06	2,118.7
64	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	5.7	24,128.51	2,214.97
64	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	5.7	24,129.47	2,139.43
64	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	5.7	24,128.83	2,255.73
64	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	5.69	24,133.94	2,229.81
64	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	5.69	24,133.64	2,171.49
64	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	5.69	24,143.05	2,258.4
64	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	5.69	24,138.91	2,045.64
64	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	5.7	24,133.01	2,212.96
256	256	256	256	10	0	$3.36 \cdot 10^7$	2.08	16.15	510.82
256	256	256	256	10	1	$3.36 \cdot 10^7$	2.07	16.22	506.97
256	256	256	256	10	2	$3.36 \cdot 10^7$	2.08	16.15	479.72
256	256	256	256	10	3	$3.36 \cdot 10^7$	2.08	16.14	490.48
256	256	256	256	10	4	$3.36 \cdot 10^7$	2.08	16.13	434.44
256	256	256	256	10	5	$3.36 \cdot 10^7$	2.08	16.1	424.31
256	256	256	256	10	6	$3.36 \cdot 10^7$	2.09	16.05	344.83
256	256	256	256	10	7	$3.36 \cdot 10^7$	2.08	16.1	372.69
256	256	256	256	10	8	$3.36 \cdot 10^7$	2.08	16.12	415.63
256	256	256	256	10	9	$3.36 \cdot 10^7$	2.08	16.15	478.86
256	1,024	1,024	1,024	10	0	$2.15 \cdot 10^9$	5.49	391.46	1,896.49
256	1,024	1,024	1,024	10	1	$2.15 \cdot 10^9$	5.49	391.25	1,669.04
256	1,024	1,024	1,024	10	2	$2.15 \cdot 10^9$	5.49	391.18	1,986.43
256	1,024	1,024	1,024	10	3	$2.15 \cdot 10^9$	5.49	391.34	1,923.14
256	1,024	1,024	1,024	10	4	$2.15 \cdot 10^9$	5.49	391.29	1,791.51
256	1,024	1,024	1,024	10	5	$2.15 \cdot 10^9$	5.49	391.43	1,868.15
256	1,024	1,024	1,024	10	6	$2.15 \cdot 10^9$	5.49	391.16	1,852.13
256	1,024	1,024	1,024	10	7	$2.15 \cdot 10^9$	5.49	391.25	1,740.72
256	1,024	1,024	1,024	10	8	$2.15 \cdot 10^9$	5.49	391.34	1,813.36
256	1,024	1,024	1,024	10	9	$2.15 \cdot 10^9$	5.48	391.54	1,806.2
256	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	5.69	24,145.75	7,457.92
256	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	5.69	24,145.74	7,164.15
256	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	5.69	24,144.37	7,288.65
256	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	5.69	24,143.12	7,471.66

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
256	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	5.69	24,147.6	7,316.29
256	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	5.69	24,144.37	7,397.38
256	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	5.69	24,146.77	7,357.63
256	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	5.69	24,149.93	7,423.96
256	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	5.69	24,145.11	7,187.65
256	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	5.69	24,145.58	7,272.14
256	16,384	16,384	16,384	10	0	$8.8 \cdot 10^{12}$	5.71	$1.54 \cdot 10^{6}$	30,705.43
256	16,384	16,384	16,384	10	1	$8.8 \cdot 10^{12}$	5.71	$1.54 \cdot 10^{6}$	30,598.66
256	16,384	16,384	16,384	10	2	$8.8 \cdot 10^{12}$	5.71	$1.54 \cdot 10^{6}$	30,745.79
256	16,384	16,384	16,384	10	3	$8.8 \cdot 10^{12}$	5.71	$1.54 \cdot 10^{6}$	30,183.49
256	16,384	16,384	16,384	10	4	$8.8 \cdot 10^{12}$	5.71	$1.54 \cdot 10^{6}$	29,838.02
256	16,384	16,384	16,384	10	5	$8.8 \cdot 10^{12}$	5.71	$1.54 \cdot 10^{6}$	30,518.09
256	16,384	16,384	16,384	10	6	$8.8 \cdot 10^{12}$	5.71	$1.54 \cdot 10^{6}$	30,164.56
256	16,384	16,384	16,384	10	7	$8.8 \cdot 10^{12}$	5.71	$1.54 \cdot 10^{6}$	29,718.9
256	16,384	16,384	16,384	10	8	$8.8 \cdot 10^{12}$	5.71	$1.54 \cdot 10^{6}$	30,055.76
256	16,384	16,384	16,384	10	9	$8.8 \cdot 10^{12}$	5.71	$1.54 \cdot 10^{6}$	30,196.71

10 Appendix C: Scale-Out Raw Results (E5-2695)

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
1	256	256	256	10	0	$3.36 \cdot 10^{7}$	0.32	106.25	0.41
1	256	256	256	10	1	$3.36 \cdot 10^{7}$	0.29	115.15	0.33
1	256	256	256	10	2	$3.36 \cdot 10^{7}$	0.26	131.26	0.52
1	256	256	256	10	3	$3.36 \cdot 10^{7}$	0.33	100.89	0.34
1	256	256	256	10	4	$3.36 \cdot 10^{7}$	0.37	90.12	0.39
1	256	256	256	10	5	$3.36 \cdot 10^{7}$	0.44	75.62	0.28
1	256	256	256	10	6	$3.36 \cdot 10^{7}$	0.34	98.82	0.23
1	256	256	256	10	7	$3.36 \cdot 10^{7}$	0.4	83.81	0.3
1	256	256	256	10	8	$3.36 \cdot 10^{7}$	0.37	90.03	0.49
1	256	256	256	10	9	$3.36 \cdot 10^{7}$	0.37	89.81	0.25
1	1,024	1,024	1,024	10	0	$2.15 \cdot 10^{9}$	0.36	5,981.55	7.82
1	1,024	1,024	1,024	10	1	$2.15 \cdot 10^{9}$	0.37	5,774.02	10.42
1	1,024	1,024	1,024	10	2	$2.15 \cdot 10^{9}$	0.36	6,001.73	8.21
1	1,024	1,024	1,024	10	3	$2.15 \cdot 10^{9}$	0.37	5,817.4	10.21
1	1,024	1,024	1,024	10	4	$2.15 \cdot 10^{9}$	0.37	5,836.46	7.24
1	1,024	1,024	1,024	10	5	$2.15 \cdot 10^{9}$	0.36	5,922.67	7.99
1	1,024	1,024	1,024	10	6	$2.15 \cdot 10^{9}$	0.37	5,850.66	5.22
1	1,024	1,024	1,024	10	7	$2.15 \cdot 10^{9}$	0.37	5,799.64	8.41
1	1,024	1,024	1,024	10	8	$2.15 \cdot 10^{9}$	0.37	5,751.82	6.66
1	1,024	1,024	1,024	10	9	$2.15 \cdot 10^{9}$	0.37	5,844.66	7.12
1	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	0.36	$3.77 \cdot 10^{5}$	148.13
1	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	0.37	$3.75 \cdot 10^{5}$	183.98
1	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	0.37	$3.76 \cdot 10^{5}$	124.29
1	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	0.36	$3.79 \cdot 10^{5}$	134.32

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
1	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	0.36	$3.78 \cdot 10^{5}$	157.89
1	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	0.36	$3.79 \cdot 10^{5}$	150.31
1	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	0.37	$3.76 \cdot 10^{5}$	139.54
1	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	0.37	$3.76 \cdot 10^{5}$	161.5
1	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	0.36	$3.77 \cdot 10^{5}$	129.06
1	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	0.36	$3.79 \cdot 10^{5}$	111.15
4	256	256	256	10	0	$3.36 \cdot 10^{7}$	1.06	31.68	9.94
4	256	256	256	10	1	$3.36 \cdot 10^{7}$	1.46	22.93	5.45
4	256	256	256	10	2	$3.36 \cdot 10^{7}$	1.06	31.7	13.6
4	256	256	256	10	3	$3.36 \cdot 10^{7}$	0.85	39.54	21.63
4	256	256	256	10	4	$3.36 \cdot 10^{7}$	0.94	35.58	16.74
4	256	256	256	10	5	$3.36 \cdot 10^{7}$	1.28	26.17	8.52
4	256	256	256	10	6	$3.36 \cdot 10^{7}$	1.71	19.6	1.51
4	256	256	256	10	7	$3.36 \cdot 10^{7}$	1.72	19.55	1.52
4	256	256	256	10	8	$3.36 \cdot 10^{7}$	1.71	19.65	1.63
4	256	256	256	10	9	$3.36 \cdot 10^{7}$	1.05	31.86	13.59
4	1,024	1,024	1,024	10	0	$2.15 \cdot 10^{9}$	1.6	1,343.47	112.7
4	1,024	1,024	1,024	10	1	$2.15 \cdot 10^{9}$	1.57	1,364.01	130.96
4	1,024	1,024	1,024	10	2	$2.15 \cdot 10^{9}$	1.58	1,358.65	116.58
4	1,024	1,024	1,024	10	3	$2.15 \cdot 10^{9}$	1.44	1,486.66	263.15
4	1,024	1,024	1,024	10	4	$2.15 \cdot 10^{9}$	1.55	1,381.21	103.92
4	1,024	1,024	1,024	10	5	$2.15 \cdot 10^{9}$	1.62	1,324.82	54.73
4	1,024	1,024	1,024	10	6	$2.15 \cdot 10^{9}$	1.62	1,325.61	71.59
4	1,024	1,024	1,024	10	7	$2.15 \cdot 10^{9}$	1.63	1,315.11	72.4
4	1,024	1,024	1,024	10	8	$2.15 \cdot 10^{9}$	1.59	1,348.57	94.32
4	1,024	1,024	1,024	10	9	$2.15 \cdot 10^{9}$	1.58	1,362.04	91.05
4	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	1.64	83,759.24	1,834.25
4	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	1.64	83,830.84	1,606.45
4	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	1.64	84,048.98	1,717.3
4	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	1.65	83,493.95	1,651.91
4	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	1.64	83,659.53	1,546.47
4	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	1.65	83,426.84	1,171.63
4	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	1.63	84,391.33	1,528.13
4	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	1.63	84,153.91	2,009.41
4	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	1.61	85,548.5	2,196.17
4	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	1.63	84,412.5	1,689.05
16	256	256	256	10	0	$3.36 \cdot 10^{7}$	5.48	6.12	9.8
16	256	256	256	10	1	$3.36 \cdot 10^{7}$	5.79	5.8	3.83
16	256	256	256	10	2	$3.36 \cdot 10^{7}$	5.65	5.94	3.61
16	256	256	256	10	3	$3.36 \cdot 10^{7}$	5.82	5.77	3.27
16	256	256	256	10	4	$3.36 \cdot 10^{7}$	5.71	5.88	3.26
16	256	256	256	10	5	$3.36 \cdot 10^{7}$	5.63	5.96	3.22
16	256	256	256	10	6	$3.36 \cdot 10^{7}$	5.8	5.78	3.11
16	256	256	256	10	7	$3.36 \cdot 10^{7}$	5.83	5.76	3.1
16	256	256	256	10	8	$3.36 \cdot 10^{7}$	5.81	5.78	3.03

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
16	256	256	256	10	9	$3.36 \cdot 10^7$	5.82	5.77	3
16	1,024	1,024	1,024	10	0	$2.15 \cdot 10^9$	6	358.1	28.32
16	1,024	1,024	1,024	10	1	$2.15 \cdot 10^9$	5.94	361.42	20.81
16	1,024	1,024	1,024	10	2	$2.15 \cdot 10^9$	6	358.02	20.68
16	1,024	1,024	1,024	10	3	$2.15 \cdot 10^9$	6	358.12	18.17
16	1,024	1,024	1,024	10	4	$2.15 \cdot 10^9$	5.98	358.99	16.77
16	1,024	1,024	1,024	10	5	$2.15 \cdot 10^9$	6	358.12	17.26
16	1,024	1,024	1,024	10	6	$2.15 \cdot 10^9$	6	358.12	16.89
16	1,024	1,024	1,024	10	7	$2.15 \cdot 10^9$	5.96	360.09	16.06
16	1,024	1,024	1,024	10	8	$2.15 \cdot 10^9$	5.99	358.81	16.03
16	1,024	1,024	1,024	10	9	$2.15 \cdot 10^9$	5.99	358.6	16.28
16	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	6.01	22,879.98	275.58
16	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	6.01	22,862.46	248.52
16	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	6.01	22,871.57	254.14
16	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	5.92	23,223.99	251.62
16	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	6	22,887.66	246.95
16	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	6.01	22,872.98	248.02
16	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	6.01	22,861.97	250.36
16	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	6.02	22,849.13	249.7
16	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	6.01	22,852.68	247.38
16	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	6.02	22,844.46	246.61
16	16,384	16,384	16,384	10	0	$8.8 \cdot 10^{12}$	5.99	$1.47 \cdot 10^6$	11,400.59
16	16,384	16,384	16,384	10	1	$8.8 \cdot 10^{12}$	5.99	$1.47 \cdot 10^6$	11,459.15
16	16,384	16,384	16,384	10	2	$8.8 \cdot 10^{12}$	5.99	$1.47 \cdot 10^6$	11,596.42
16	16,384	16,384	16,384	10	3	$8.8 \cdot 10^{12}$	5.99	$1.47 \cdot 10^6$	11,409.82
16	16,384	16,384	16,384	10	4	$8.8 \cdot 10^{12}$	6	$1.47 \cdot 10^6$	11,369.89
16	16,384	16,384	16,384	10	5	$8.8 \cdot 10^{12}$	5.99	$1.47 \cdot 10^6$	11,371
16	16,384	16,384	16,384	10	6	$8.8 \cdot 10^{12}$	5.99	$1.47 \cdot 10^6$	11,411.25
16	16,384	16,384	16,384	10	7	$8.8 \cdot 10^{12}$	5.99	$1.47 \cdot 10^6$	11,407.13
16	16,384	16,384	16,384	10	8	$8.8 \cdot 10^{12}$	5.99	$1.47 \cdot 10^6$	11,423.57
16	16,384	16,384	16,384	10	9	$8.8 \cdot 10^{12}$	6	$1.47 \cdot 10^6$	11,285.03
64	256	256	256	10	0	$3.36 \cdot 10^7$	2.88	11.64	75.97
64	256	256	256	10	1	$3.36 \cdot 10^7$	2.9	11.57	247.49
64	256	256	256	10	2	$3.36 \cdot 10^7$	2.9	11.55	245.83
64	256	256	256	10	3	$3.36 \cdot 10^7$	2.86	11.74	246.99
64	256	256	256	10	4	$3.36 \cdot 10^7$	2.9	11.56	246.6
64	256	256	256	10	5	$3.36 \cdot 10^7$	2.9	11.57	98.47
64	256	256	256	10	6	$3.36 \cdot 10^7$	2.9	11.57	98.44
64	256	256	256	10	7	$3.36 \cdot 10^7$	2.9	11.57	98.71
64	256	256	256	10	8	$3.36 \cdot 10^7$	2.86	11.72	99.17
64	256	256	256	10	9	$3.36 \cdot 10^7$	2.9	11.57	246.06
64	1,024	1,024	1,024	10	0	$2.15 \cdot 10^9$	21.28	100.9	579.01
64	1,024	1,024	1,024	10	1	$2.15 \cdot 10^9$	21.26	101.03	864.89
64	1,024	1,024	1,024	10	2	$2.15 \cdot 10^9$	21.21	101.26	861.54
64	1,024	1,024	1,024	10	3	$2.15 \cdot 10^9$	21.29	100.89	843.4

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
64	1,024	1,024	1,024	10	4	$2.15 \cdot 10^9$	21.29	100.89	844.95
64	1,024	1,024	1,024	10	5	$2.15 \cdot 10^9$	21.15	101.53	848.09
64	1,024	1,024	1,024	10	6	$2.15 \cdot 10^9$	21.32	100.74	823.32
64	1,024	1,024	1,024	10	7	$2.15 \cdot 10^9$	21.22	101.21	838.28
64	1,024	1,024	1,024	10	8	$2.15 \cdot 10^9$	21.42	100.24	882.28
64	1,024	1,024	1,024	10	9	$2.15 \cdot 10^9$	21.29	100.87	840.42
64	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	23.98	5,732.25	1,140.01
64	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	23.98	5,730.39	1,984.33
64	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	23.93	5,743.27	1,937.15
64	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	23.93	5,744.27	1,937.83
64	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	23.84	5,765.81	1,945.29
64	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	23.81	5,772.79	2,023.14
64	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	23.81	5,771.55	1,926.5
64	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	23.75	5,786.39	1,971.01
64	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	23.7	5,799.07	2,044.53
64	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	23.74	5,790.17	1,941.88
64	16,384	16,384	16,384	10	0	$8.8 \cdot 10^{12}$	22.65	$3.88 \cdot 10^5$	19,205.81
64	16,384	16,384	16,384	10	1	$8.8 \cdot 10^{12}$	22.59	$3.89 \cdot 10^5$	23,475.97
64	16,384	16,384	16,384	10	2	$8.8 \cdot 10^{12}$	22.6	$3.89 \cdot 10^5$	23,651.57
64	16,384	16,384	16,384	10	3	$8.8 \cdot 10^{12}$	22.58	$3.9 \cdot 10^5$	23,782.32
64	16,384	16,384	16,384	10	4	$8.8 \cdot 10^{12}$	22.62	$3.89 \cdot 10^5$	24,708.92
64	16,384	16,384	16,384	10	5	$8.8 \cdot 10^{12}$	22.57	$3.9 \cdot 10^5$	23,955.09
64	16,384	16,384	16,384	10	6	$8.8 \cdot 10^{12}$	22.62	$3.89 \cdot 10^5$	23,006.49
64	16,384	16,384	16,384	10	7	$8.8 \cdot 10^{12}$	22.64	$3.89 \cdot 10^5$	24,400.9
64	16,384	16,384	16,384	10	8	$8.8 \cdot 10^{12}$	22.6	$3.89 \cdot 10^5$	24,456.82
64	16,384	16,384	16,384	10	9	$8.8 \cdot 10^{12}$	22.61	$3.89 \cdot 10^5$	24,002.73
256	256	256	256	10	0	$3.36 \cdot 10^7$	1.05	31.98	1,334.79
256	256	256	256	10	1	$3.36 \cdot 10^7$	1.04	32.26	2,154.73
256	256	256	256	10	2	$3.36 \cdot 10^7$	1.03	32.61	1,994.15
256	256	256	256	10	3	$3.36 \cdot 10^7$	1.05	31.88	1,983.65
256	256	256	256	10	4	$3.36 \cdot 10^7$	1.05	31.95	1,990.95
256	256	256	256	10	5	$3.36 \cdot 10^7$	1.04	32.19	2,006.99
256	256	256	256	10	6	$3.36 \cdot 10^7$	1.05	31.82	2,475.74
256	256	256	256	10	7	$3.36 \cdot 10^7$	1.05	32.06	2,617.51
256	256	256	256	10	8	$3.36 \cdot 10^7$	1.06	31.69	2,146.51
256	256	256	256	10	9	$3.36 \cdot 10^7$	1.06	31.66	1,987.36
256	1,024	1,024	1,024	10	0	$2.15 \cdot 10^9$	37.95	56.59	2,967.75
256	1,024	1,024	1,024	10	1	$2.15 \cdot 10^9$	39.03	55.03	2,934.17
256	1,024	1,024	1,024	10	2	$2.15 \cdot 10^9$	39.09	54.93	2,944.91
256	1,024	1,024	1,024	10	3	$2.15 \cdot 10^9$	39.07	54.96	3,033.53
256	1,024	1,024	1,024	10	4	$2.15 \cdot 10^9$	38.15	56.29	2,757.6
256	1,024	1,024	1,024	10	5	$2.15 \cdot 10^9$	38.62	55.61	3,128.82
256	1,024	1,024	1,024	10	6	$2.15 \cdot 10^9$	39.09	54.94	3,098.18
256	1,024	1,024	1,024	10	7	$2.15 \cdot 10^9$	39.14	54.87	3,052.84
256	1,024	1,024	1,024	10	8	$2.15 \cdot 10^9$	38.11	56.35	3,019.93

p	m	q	n	trials	trial	flops	GF/s	tmult	tcomm
256	1,024	1,024	1,024	10	9	$2.15 \cdot 10^9$	38.74	55.43	2,673
256	4,096	4,096	4,096	10	0	$1.37 \cdot 10^{11}$	93.1	1,476.32	3,882.7
256	4,096	4,096	4,096	10	1	$1.37 \cdot 10^{11}$	92.77	1,481.56	6,168.18
256	4,096	4,096	4,096	10	2	$1.37 \cdot 10^{11}$	93.57	1,468.87	5,846.41
256	4,096	4,096	4,096	10	3	$1.37 \cdot 10^{11}$	92.51	1,485.7	5,806.1
256	4,096	4,096	4,096	10	4	$1.37 \cdot 10^{11}$	92.57	1,484.64	5,591.05
256	4,096	4,096	4,096	10	5	$1.37 \cdot 10^{11}$	92.44	1,486.72	5,719.98
256	4,096	4,096	4,096	10	6	$1.37 \cdot 10^{11}$	92.83	1,480.58	6,025.78
256	4,096	4,096	4,096	10	7	$1.37 \cdot 10^{11}$	92.39	1,487.57	6,645.81
256	4,096	4,096	4,096	10	8	$1.37 \cdot 10^{11}$	92.38	1,487.75	5,943.78
256	4,096	4,096	4,096	10	9	$1.37 \cdot 10^{11}$	92.58	1,484.61	6,144.75
256	16,384	16,384	16,384	10	0	$8.8 \cdot 10^{12}$	92.13	95,477.84	15,049.62
256	16,384	16,384	16,384	10	1	$8.8 \cdot 10^{12}$	89.95	97,789.49	26,098.9
256	16,384	16,384	16,384	10	2	$8.8 \cdot 10^{12}$	88.97	98,867.89	27,549.13
256	16,384	16,384	16,384	10	3	$8.8 \cdot 10^{12}$	88.4	99,503.8	27,650.32
256	16,384	16,384	16,384	10	4	$8.8 \cdot 10^{12}$	87.81	$1 \cdot 10^5$	28,393.98
256	16,384	16,384	16,384	10	5	$8.8 \cdot 10^{12}$	87.01	$1.01 \cdot 10^5$	30,098.74
256	16,384	16,384	16,384	10	6	$8.8 \cdot 10^{12}$	88.58	99,300.41	27,146.28
256	16,384	16,384	16,384	10	7	$8.8 \cdot 10^{12}$	88.44	99,459.33	29,092.02
256	16,384	16,384	16,384	10	8	$8.8 \cdot 10^{12}$	88.43	99,465.86	28,129.62
256	16,384	16,384	16,384	10	9	$8.8 \cdot 10^{12}$	87.87	$1 \cdot 10^5$	29,308.97

References

[1] *Beamer (LaTeX)*. Wikimedia Foundation, Inc. Nov. 23, 2017. URL: https://en.wikipedia.org/wiki/Beamer%5C_(LaTeX) (visited on 12/07/2017).

[2] Johannes Braams. *The alltt environment*. June 16, 1997. URL: http://mirrors.ctan.org/macros/latex/base/alltt.pdf (visited on 12/02/2017).

[3] *Bridges User Guide. Using Bridges' GPU nodes*. Pittsburgh Supercomputing Center. Oct. 31, 2017. URL: https://www.psc.edu/bridges/user-guide/gpu-use (visited on 12/04/2017).

[4] *Cannon's Algorithm*. Wikimedia Foundation, Inc. Nov. 28, 2017. URL: https://en.wikipedia.org/wiki/Cannon%5C%27s%5C_algorithm (visited on 11/30/2017).

[5] David Carlisle. *The enumerate package*. July 23, 2015. URL: http://mirrors.ctan.org/macros/latex/required/tools/enumerate.pdf (visited on 11/23/2017).

[6] *cblas_?gemm*. Intel Corporation. URL: https://software.intel.com/en-us/mkl-developer-reference-c-cblas-gemm (visited on 12/02/2017).

[7] *cblas_sgemmx.c*. Intel Corporation. URL: https://software.intel.com/en-us/mkl-developer-reference-c-cblas%5C_sgemmx (visited on 12/02/2017).

[8] Tim Child. *How to optimize matrix multiplication using OpenACC?* Ed. by Mark Harris. Sept. 14, 2012. URL: https://stackoverflow.com/questions/11791843/how-to-optimize-matrix-multiplication-using-openacc (visited on 12/06/2017).

[9] Mat Colgrove. *OpenACC parallel kernels not getting generated*. Jan. 28, 2016. URL: https://stackoverflow.com/questions/35035427/openacc-parallel-kernels-not-getting-generated (visited on 12/06/2017).

[10] Alexander Feder. *BibTex Format Description*. 2006. URL: http://www.bibtex.org/Format/ (visited on 10/14/2017).

[11] Alexander Feder. *How to use BibTeX*. 2006. URL: http://www.bibtex.org/Using/ (visited on 10/14/2017).

[12] Christian Feuersänger. *Manual for Package Pgfplots*. June 5, 2017. URL: http://mirrors.ctan.org/graphics/pgf/contrib/pgfplots/doc/pgfplots.pdf.

[13] Christian Feuersänger. *Manual for Package PgfplotsTable*. June 5, 2017. URL: http://mirrors.ctan.org/graphics/pgf/contrib/pgfplots/doc/pgfplotstable.pdf.

[14] *GCC Wiki. OpenACC*. Free Software Foundation, Inc. July 27, 2017. URL: https://gcc.gnu.org/wiki/OpenACC (visited on 12/04/2017).

[15] H. Gupta and P. Sadayappan. *Communication Efficient Matrix-Multiplication on Hypercubes*. 1994. URL: http://ilpubs.stanford.edu:8090/59/1/1994-25.pdf (visited on 11/13/2017).

[16] Lennart Johnsson. *Introduction to HPC Lecture 10. Performance and Parallel Computing Concepts*. Blackboard. Lecture Slides. Department of Computer Science, University of Houston, Aug. 24, 2017.

[17] Lennart Johnsson. *Introduction to HPC Lecture 17. Parallel Algorithms: Matrix Multiplication*. Department of Computer Science, University of Houston. Oct. 24, 2017.

[18] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd ed. Prentice Hall Software Series. Englewood Cliffs, New Jersey: Prentice Hall PTR, 1988.

[19] Philip Kime and François Charette. *biber. A backend bibliography processor for biblatex*. Dec. 5, 2016. URL: http://mirrors.ctan.org/biblio/biber/documentation/biber.pdf.

[20] *LATEX 2 for authors*. LATEX3 Project Team. July 31, 2001. URL: http://www.latex-project.org/help/documentation/usrguide.pdf.

[21] *LaTeX/Customizing Page Headers and Footers*. The Wikimedia Project. Oct. 4, 2017. URL: https://en.wikibooks.org/wiki/LaTeX/Customizing%5C_Page%5C_Headers%5C_and%5C_Footers.

[22] *LaTeX/List Structures*. The Wikimedia Project. Oct. 4, 2017. URL: https://en.wikibooks.org/wiki/LaTeX/List%5C_Structures.

[23] *LaTeX/Mathematics*. The Wikimedia Project. Oct. 4, 2017. URL: https://en.wikibooks.org/wiki/LaTeX/Mathematics.

[24] Philip Lehman et al. *The biblatex Package*. Nov. 16, 2016. URL: http://mirrors.ctan.org/macros/latex/contrib/biblatex/doc/biblatex.pdf.

[25] *MPI: A Message-Passing Interface Standard*. Version 3.1. Message Passing Interface Forum. June 4, 2015. URL: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[26] *NVIDIA® Tesla® P100 GPU Accelerator*. NVIDIA Corporation. Oct. 6, 2016. URL: https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf (visited on 11/13/2017).

[27] *OpenMP Application Programming Interface*. Version 4.5. OpenMP Architecture Review Board. Nov. 2015. URL: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[28] Geoffrey M. Poore. *The minted package. Highlighted source code in LATEX*. July 19, 2017. URL: http://mirrors.ctan.org/macros/latex/contrib/minted/minted.pdf.

[29] *Single-Precision Floating Point Format. Precision limits on integer values*. Wikimedia Foundation, Inc. Nov. 28, 2017. URL: https://en.wikipedia.org/wiki/Single-precision%5C_floating-point%5C_format%5C#Precision%5C_limits%5C_on%5C_integer%5C_values (visited on 11/30/2017).

[30] Ryan Smith. *NVIDIA Launches Tesla K80, GK210 GPU*. Nov. 14, 2017. URL: https://www.anandtech.com/show/8729/nvidia-launches-tesla-k80-gk210-gpu (visited on 11/13/2017).

[31] Nadir Soualem. *How to make a presentation with Latex - Introduction to Beamer*. July 14, 2007. URL: https://math-linux.com/latex-26/article/how-to-make-a-presentation-with-latex-introduction-to-beamer (visited on 12/07/2017).

[32] *Stampede2 User Guide*. Texas Advanced Supercomputing Center, University of Texas at Austin. Oct. 22, 2017. URL: https://portal.tacc.utexas.edu/user-guides/stampede2 (visited on 11/13/2017).

[33] *The OpenACC(R) Application Programming Interface*. Version 2.6. OpenACC-Standard.org. Nov. 6, 2017. URL: https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf (visited on 12/04/2017).

[34] M. Tibbits. *How can I split a beamer bibliography across two slides?* June 14, 2011. URL: https://tex.stackexchange.com/questions/20660/how-can-i-split-a-beamer-bibliography-across-two-slides (visited on 12/07/2017).

[35] *Understanding How to Have One Legend for a Group Plot*. June 22, 2016. URL: https://tex.stackexchange.com/questions/315987/understanding-how-to-have-one-legend-for-a-group-plot (visited on 12/12/2017).

[36] Michael Wolfe. *OpenACC 2.0 and the PGI Accelerator Compiler*. Mar. 20, 2013. URL: http://on-demand.gputechconf.com/gtc/2013/presentations/S3447-OpenACC-2%5C%20-PGI-Accelerator-Compilers.pdf (visited on 12/09/2017).