Cannon's Algorithm for Matrix Multiplication Scale Up or Scale Out?

Michael Yantosca

University of Houston

December 7, 2017

Scale-Up vs. Scale-Out

To explore the relative merits between single-node computer power in depth versus multi-node compute power in breadth, I propose an experiment comparing the performance and efficiency of Cannon's algorithm[4, 3] for the matrix multiplication problem $C = A \times B$ between a single-node OpenMP implementation utilizing accelerators and a multi-node MPI implementation spread across a cluster.

Data

- C-stationary case
- Limited number of matrix sizes
  - 256x256
  - 1024x1024
  - 4096x4096
  - 16384x16384
  - 256x1024
  - 256x4096
  - 256x16384
  - 16384x256
  - 16384x256
  - 4096x256
  - 1024x256
- Performance trials: random numbers
- Validation trials: specially devised A and B so each cell of product C has a unique value

Resources

- Scale-Up
  - OpenMP offloading to GPU nodes on BRIDGES
  - backup: OpenACC offloading to GPU nodes on BRIDGES
  - backup to the backup: CUDA offloading to GPU nodes on BRIDGES
  - if time permits: OpenMP on KNL nodes on STAMPEDE2
- Scale-Out
  - MPI distribution over non-accelerated nodes on BRIDGES

Efficiency: Scale-Out (MPI)

- Strong-scaling roofline model
- Basis: single non-GPU node on BRIDGES
  - 2.30 GHz Intel E5-2695 (28-cores)
  - 2.30 GHz * 28 cores * 4 SIMD instructions/cycle (AVX256) = 257.6 GFLOPs/s/node
  - Empirical reference: MKL `cblas_sgemm`

Efficiency: Scale-Up (OpenMP/OpenACC)

- accelerator roofline based on published specs
- NVIDIA P100
  - 9.3 SP TFLOPs/s/card[7]
- NVIDIA K80
  - 8.74 TFLOPs/s/card[10]
- Empirical reference: tweaked matrixMulCUBLAS sample provided with CUDA toolkit
- KNL (STAMPEDE2)
  - 1.4 GHz * 68 cores * 8 SIMD instructions/cycle (AVX512) = 761.6 GFLOPs/s/node[11][1]

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

Hypotheses

- OpenMP[8]/OpenACC[12] solution will outperform the MPI solution for smaller matrix sizes.
- Scale-Out will outperform Scale-Up when local contention for resources exceeds the communication overhead across nodes.
- The inflection point will depend on the hierarchical layout of the accelerator in use.
    - Any loss of data parallelism where the accelerator has to run sequential loops will kill performance.

Validation: The Goal

- First priority is to catch logic errors early.
- Exacting constraints
  - Each cell of C has a unique value.
  - Per-cell validation must run in constant time.
    - Ideally, $f(i,j) = \sum_{k=1}^{N} A_{i,k} \cdot B_{k,j}$.
  - The validation should protect against row and column drift.
- Inspiration: $iq + j$ (C 2D array index offset calc[5, p. 113])

Validation: A Cautionary Tale

- Spent an inordinate amount of time wrestling with the algebra.
- Refined solution
  - A populated with row index i
  - $B = [\mathbf{1}\ \mathbf{1}\ \ldots\ \mathbf{1}]$
  - $C_0 = [\mathbf{0}\ \mathbf{1}\ \ldots\ \mathbf{n}]$
  - $C = A \times B + C_0$
  - C populated with $iq + j$
- Good?

Validation: A Cautionary Tale

- B is a homogeneous field of ones.
- No protection against column drift!
- Refined solution
  - A populated with $(i + 1)$
  - $B = [\mathbf{0}\ \mathbf{1}\ \ldots\ \mathbf{n}]$
  - C_0 populated with $-((q-1)j + (j-1)iq)$
  - $C = A \times B + C_0$
  - C populated with $iq + j$
- Good?

Validation: A Cautionary Tale

- Single-precision floating point exact integer precision range: $[-2^{24}, 2^{24}]$[9]
- Method requires upper bound of $16383 \times 16383 + 16382 \times 16383 \times 16384$, i.e., something on the order of 2^{42}
- Attempts to compress this range by reducing i and j failed miserably.
- Fidelity loss in fractions as bad as loss between large numbers.

Validation: The Outcome

- Fell back on $A = \mathbf{1}$, $B = \mathbf{1}$, $C_0 = \mathbf{0}$
- Checked that $C = \mathbf{q}$
- Did manual tests with small matrices on Cannon's algorithm implementations.
  - e.g., A and B populated with $iq + j$ for 2x2 and 4x4 matrices

Scale-Up: Implementation

- Started with OpenMP implementation
- No support for GPU offloading via OpenMP on BRIDGES, even with gcc-7.2
- Ported implementation to OpenACC
- Had to use PGI compiler
  - BRIDGES supports PGI[1]
  - gcc 7 has no support for nested acc loops or host fallback![2]
- Missed adding `-acc` flag to `Makefile`
  - Wasn't actually offloading to GPU
  - Finally got it to compile and loops could not be parallelized.
    - Dependency on loop variables for matrix offset indexing.
  - Worked out remaining bugs and got a slight speedup compared to previous.
  - But...is this really Cannon's algorithm?

Scale-Up: PGI Compiler Output

```
make[1]: Entering directory '/home/michael/cosc6365/final/src'
main:
215, Loop is parallelizable
217, Loop is parallelizable
219, Loop is parallelizable
Accelerator kernel generated
Generating Tesla code
i 215, #pragma acc loop gang /* blockIdx.z */
x 217, #pragma acc loop gang /* blockIdx.y */
y 219, #pragma acc loop gang /* blockIdx.x */
k 224, #pragma acc loop seq
j 226, #pragma acc loop vector(128) /* threadIdx.x */
224, Loop is parallelizable
226, Loop is parallelizable
make[1]: Leaving directory '/home/michael/cosc6365/final/src'
```

Scale-Up: Algorithmic Notes

- Device matrix dA allocates an extra block column for temporarily storing wraparound rotation.
- Device matrix dB allocates an extra block row for temporarily storing wraparound rotation.
- The rotation loop iterates one more than the corresponding processor grid dimension.
- Host matrices A, B, and C and device matrices dA, dB and dC are allocated as single blocks.
  - Subarrays are accessed by calculating the appropriate row-major offset.
  - NB: Processor grid dimensions x and y start at 1.
- Local multiplication is implemented as a typical ikj multiplication.

Scale-Out: Implementation

- Started with port of OpenACC scale-up implementation to MPI framework.
- Employed RMA communication between nodes for initial shearing and cyclic rotation.
- Synchronization is Post-Start-Complete-Wait[6, pp. 456-463].

Scale-Out: Phase 0 (Allocation)

- Root process $P_{0,0}$ fills A and B and zeroes out C.
- Root process $P_{0,0}$ sets up communication windows for A, B, and C
- Non-root processes $P_{x\neq 0,y\neq 0}$ set up null communication windows for A, B, and C.
- All processes $P_{x,y}$ allocate dA, dB, and dC and set up communication windows for dA and dB.[2]

[2] dA and dB are allotted double the required space to provide a ghost block/send buffer to avoid corruption during cyclic rotation.

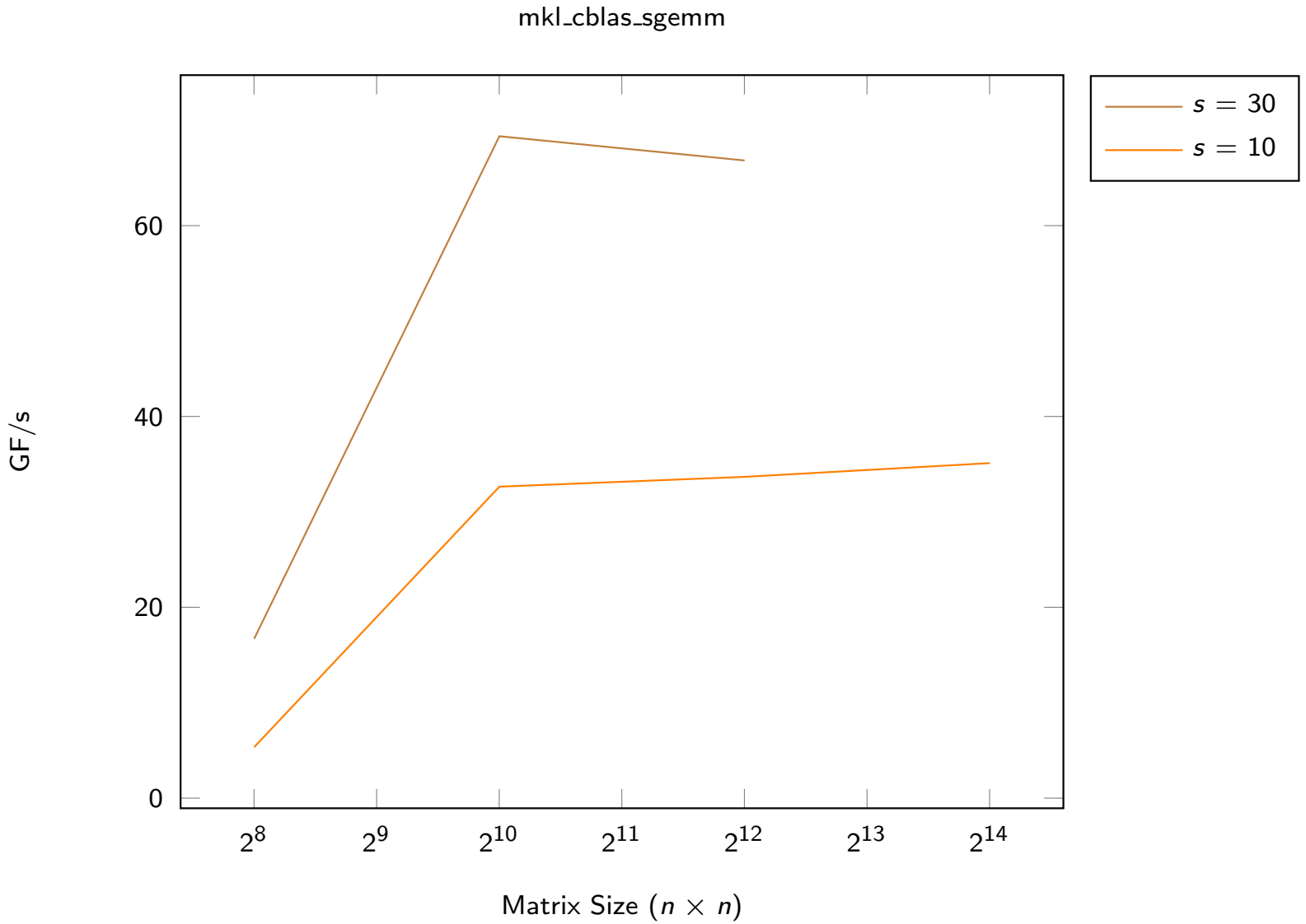
Scale-Out: Phase 1 (Shearing)

- Root process $P_{0,0}$ issues `MPI_Win_post` call to `MPI_COMM_WORLD`.
- All processes $P_{x,y}$ issue `MPI_Win_start`, `MPI_Get`, and `MPI_Win_complete` for $A_{x,(y+x) \bmod c}$, $B_{(x+y) \bmod b,y}$, and $C_{x,y}$.
- Root process $P_{0,0}$ issues `MPI_Win_wait`.

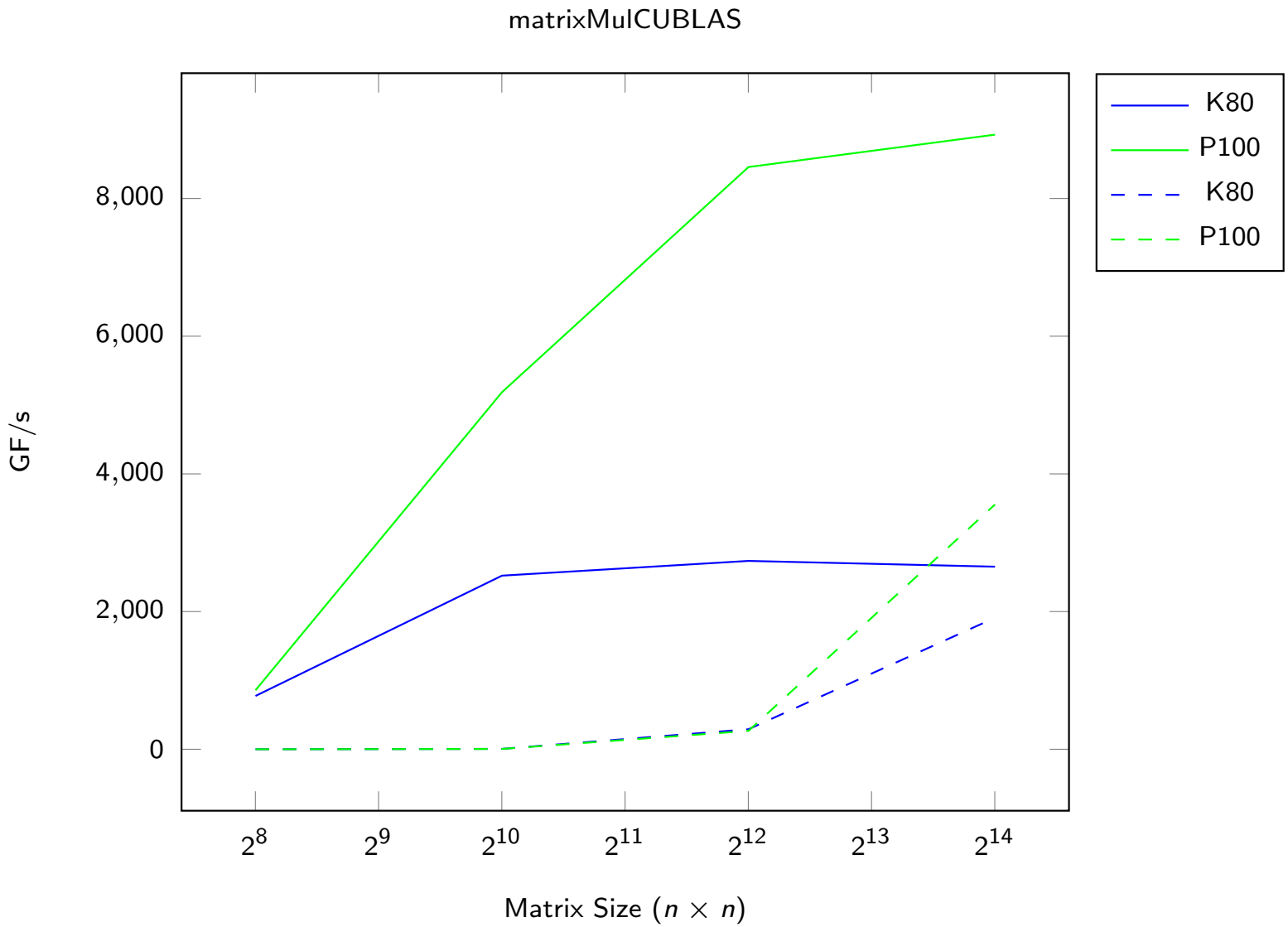
Scale-Out: Phase 2 (Rotate-Multiply-Add)

- Loop until $P_{x,y}$ has processed each row x and column y at most once.
  - Cyclic rotation
    - $P_{x,y}$ copies its present copies of dA and dB to the respective ghost blocks.
    - $P_{x,y}$ issues corresponding `MPI_Win_post` calls to its east and south neighbors.
    - $P_{x,y}$ issues `MPI_Win_start` calls on dA and dB to its west and north neighbors, respectively.
    - $P_{x,y}$ sends its present copies of dA and dB to its west and north neighbors, respectively.
    - $P_{x,y}$ awaits new copies of dA and dB from its east and south neighbors, respectively.
  - All processes $P_{x,y}$ perform local ikj multiplication.

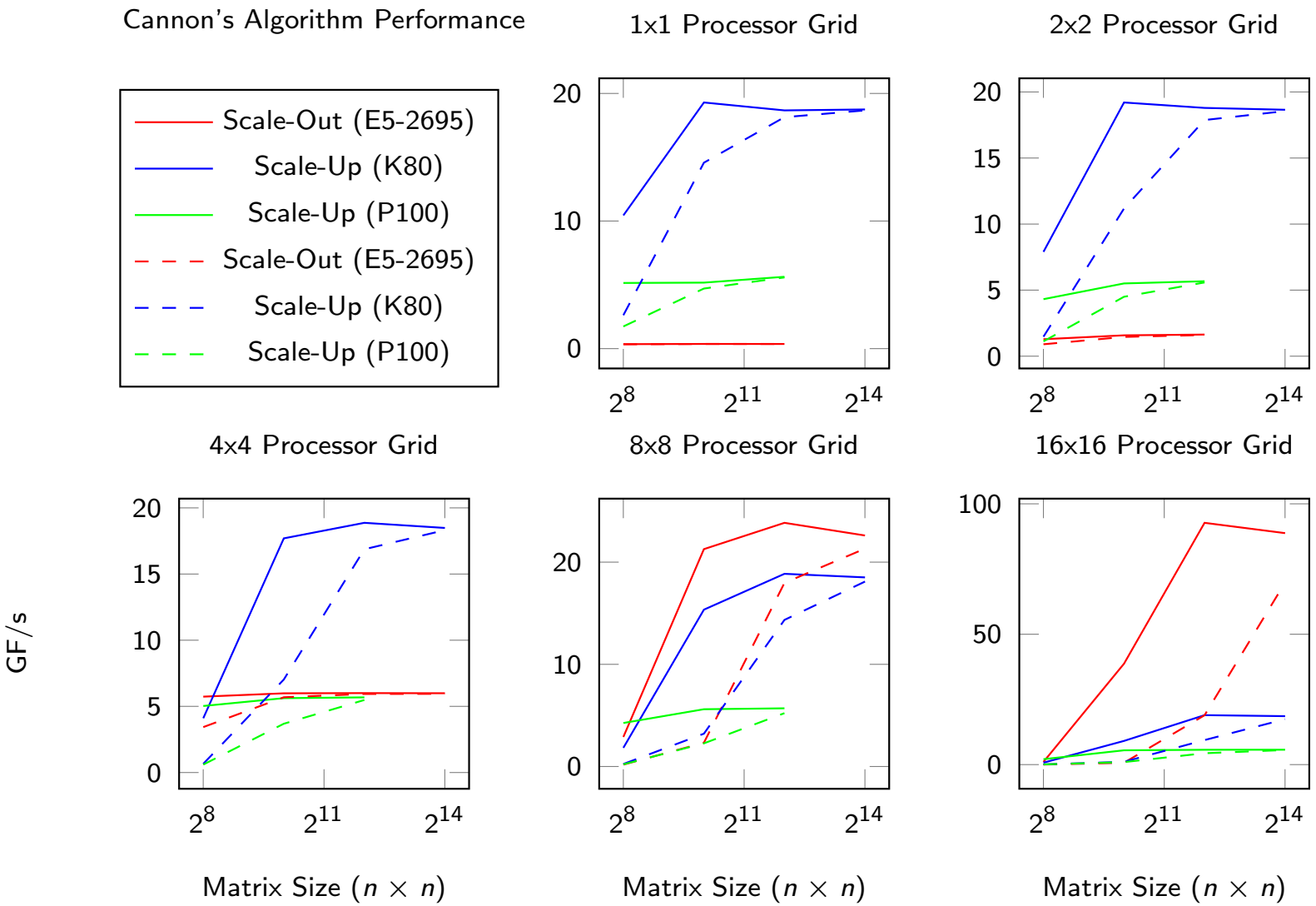
MKL Results (Performance)



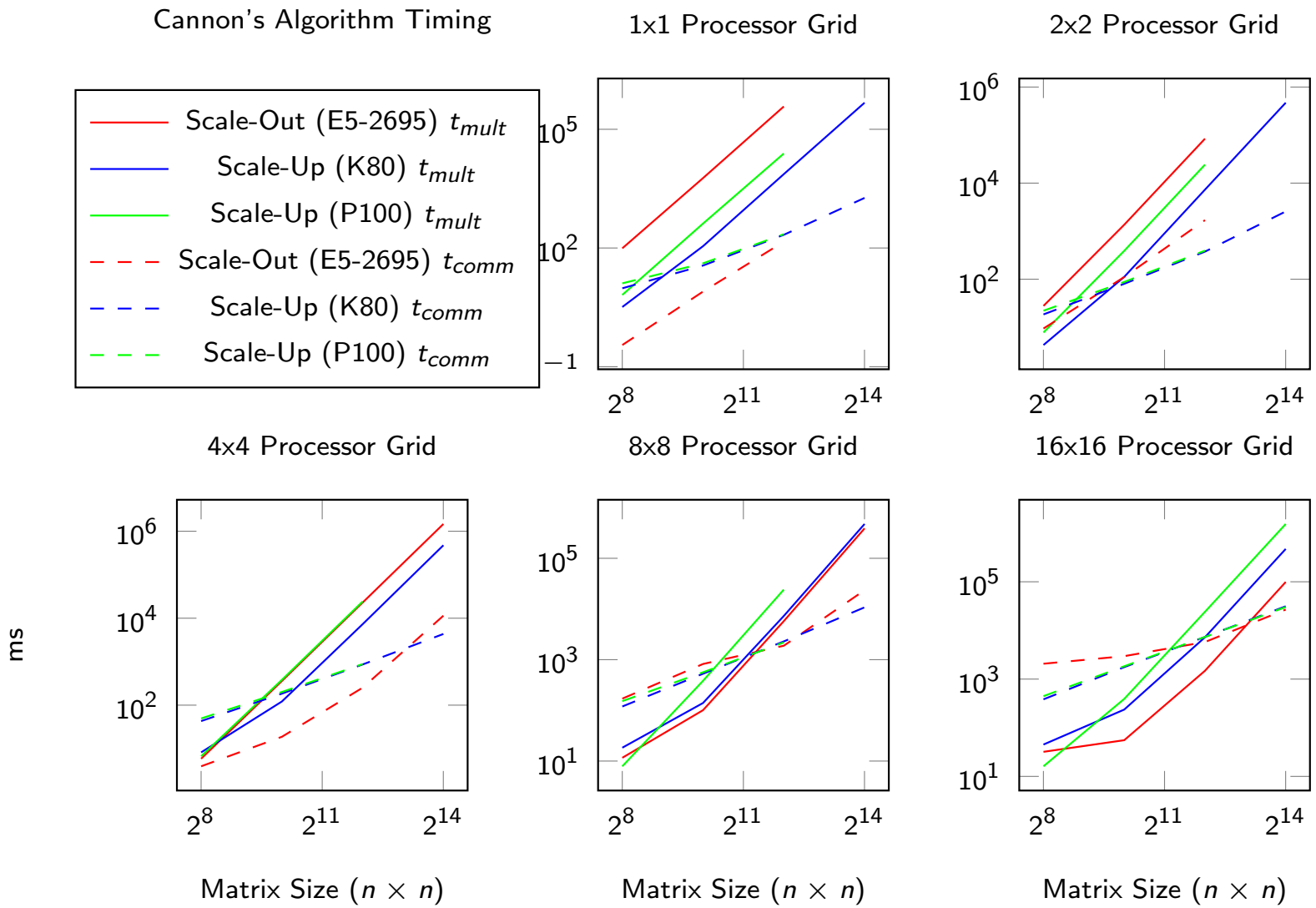
CUBLAS Results (Performance)



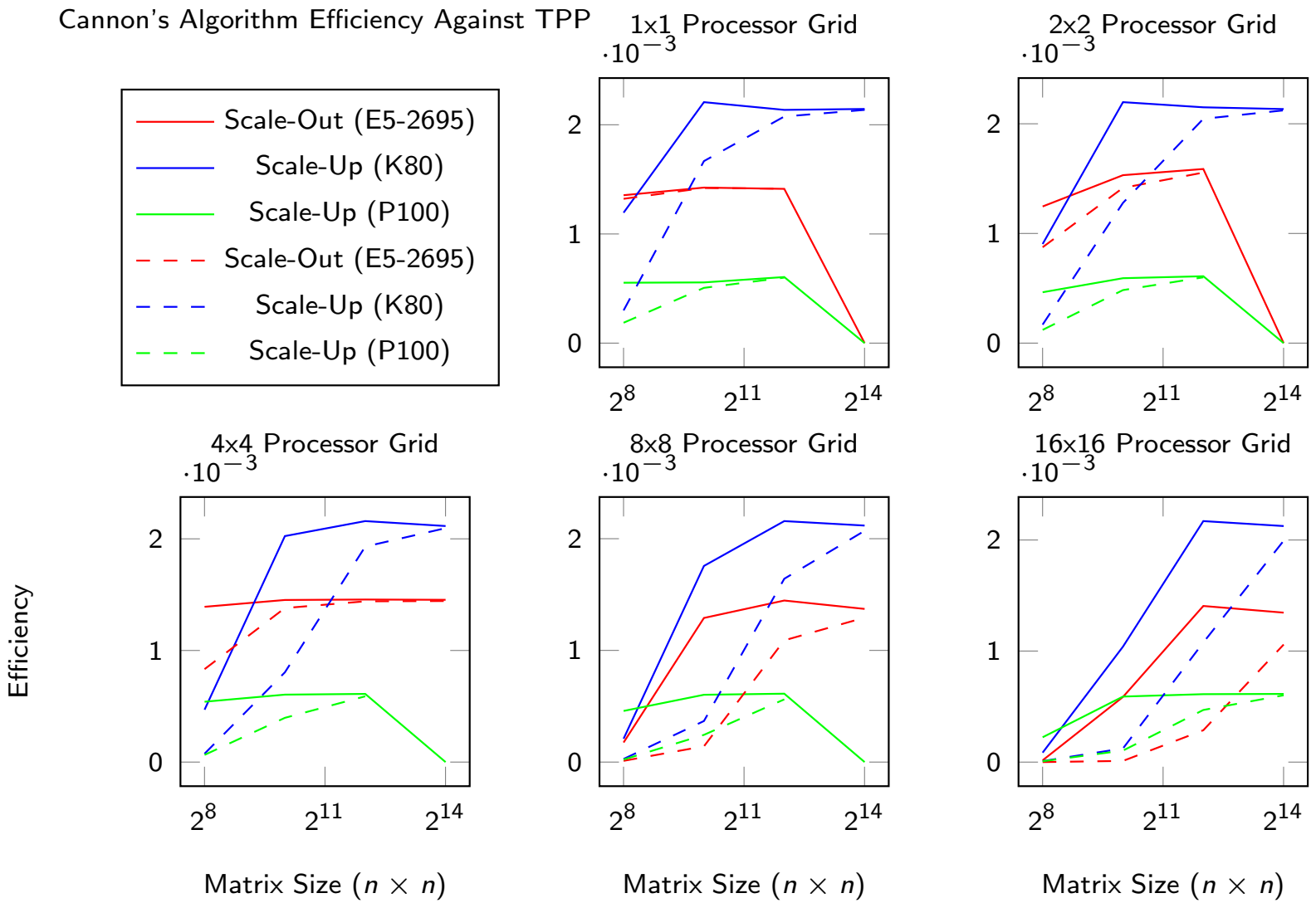
Cannon's Algorithm Results (Performance)



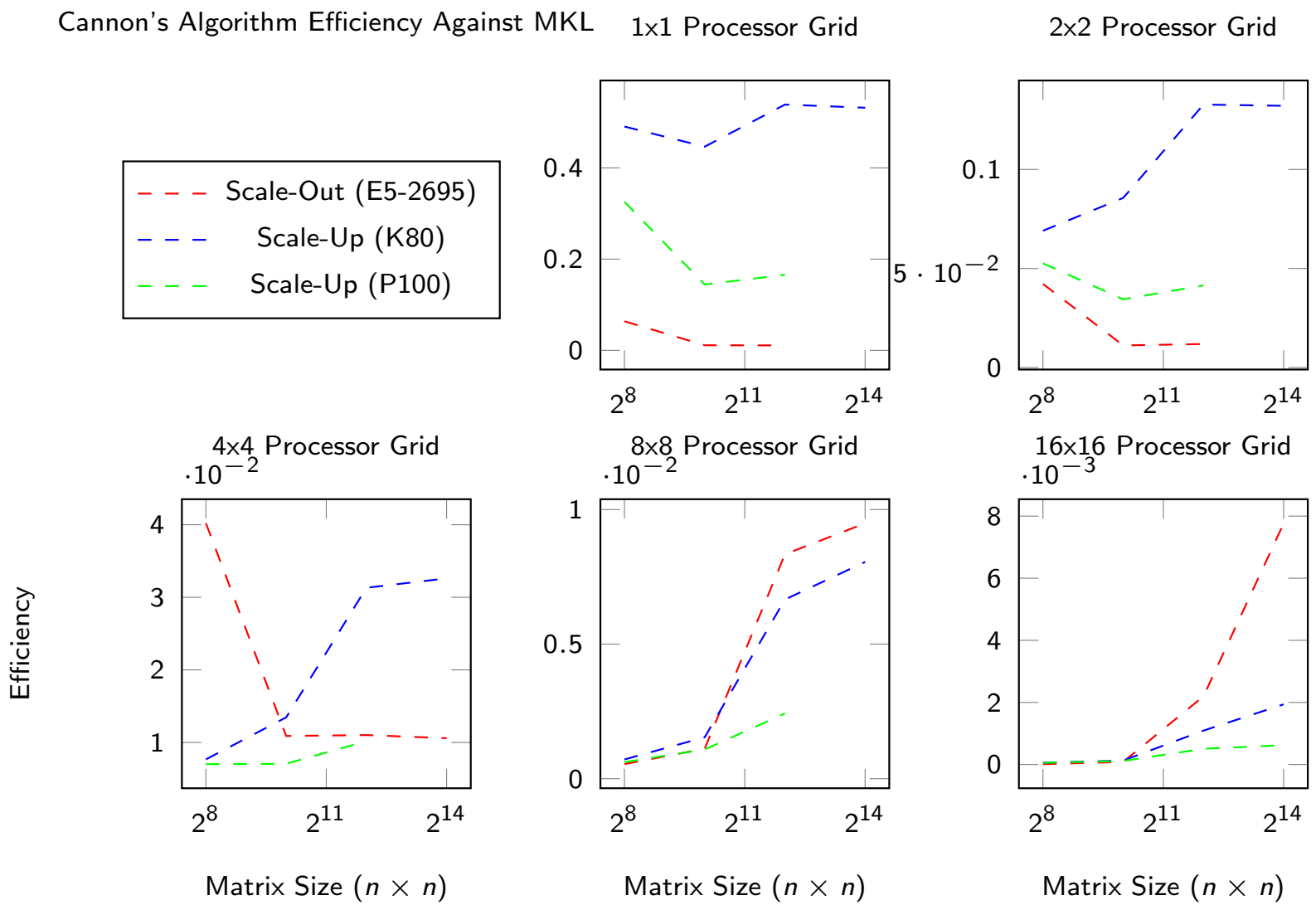
Cannon's Algorithm Results (Timing)



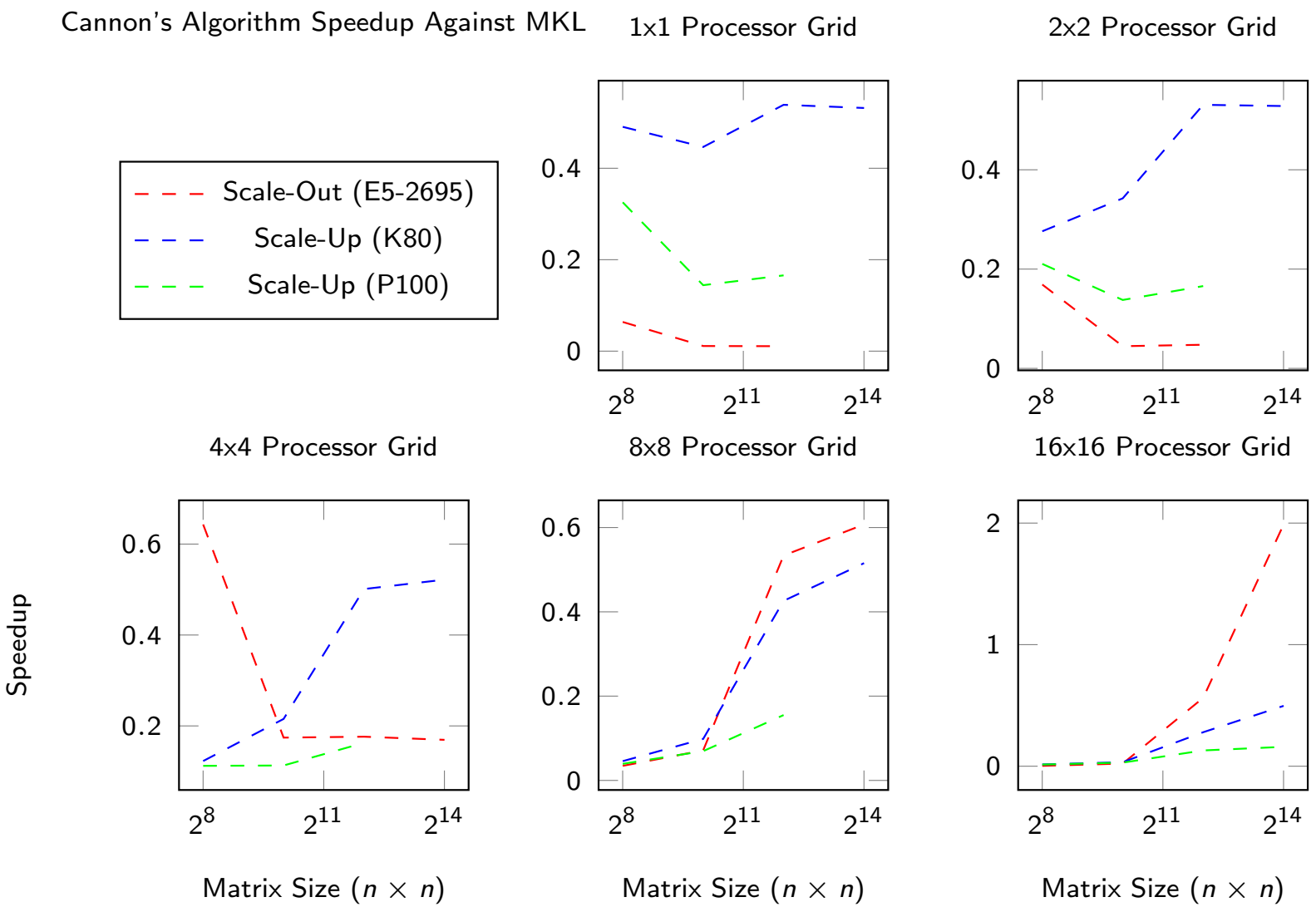
Cannon's Algorithm Results (Efficiency:TPP)



Cannon's Algorithm Results (Efficiency:MKL)



Cannon's Algorithm Results (Speedup:MKL)



Conclusions

- Don't get stuck in the realm of sunk-cost fallacy.
  - Know when to cut losses and move on.
- Local multiplication makes it extremely difficult to generalize Cannon's algorithm to non-square matrices.
- Cannon's algorithm seems better suited for scale-out implementations. Use CUBLAS for GPU.
- The communication cost makes implementing the algorithm difficult to justify unless node communications are implemented in a supremely efficient manner.

References I


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).


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


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).


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

References II

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.

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`.

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).

References III

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.

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).

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).

References IV

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).

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).