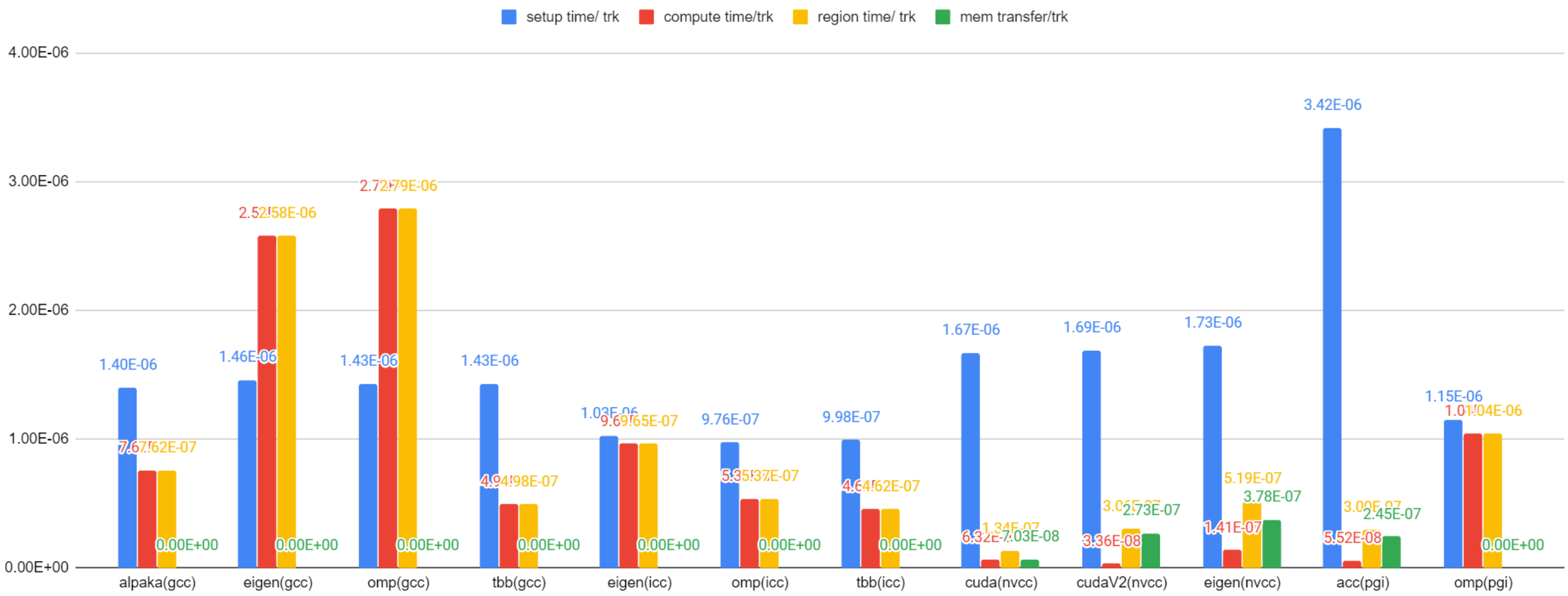
Propagate to Z Summary

Tres Reid

Cornell University

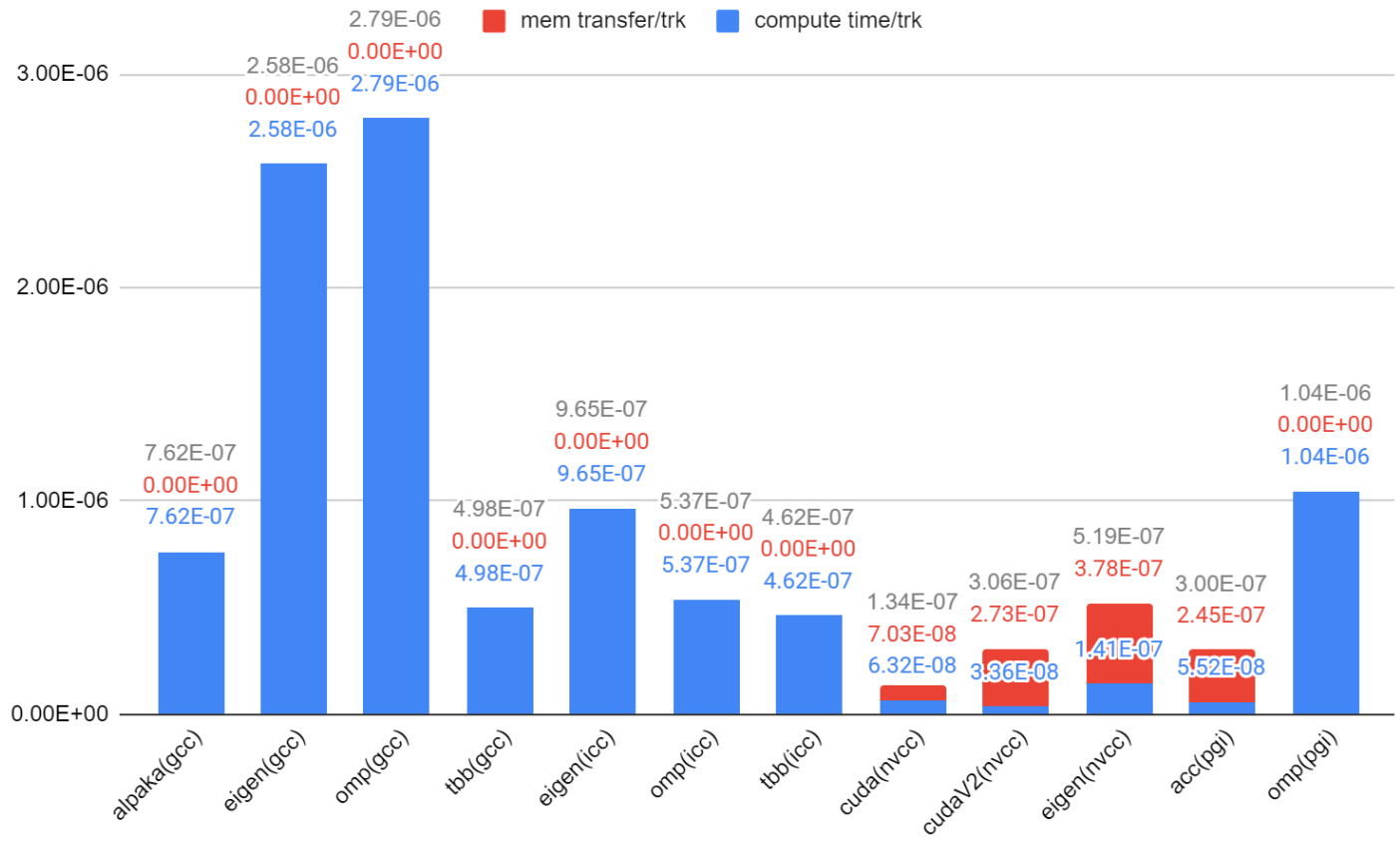
4/28/2020

Summary of Results



Summary of Results

- GPU versions are consistently faster than CPU versions even when including memory transfer times
  - Fastest version seems to be CUDA.
  - Acc has a faster compute time but a slower memory transfer than CUDA
- Tbb remains about the same regardless of the compiler despite omp being compiler dependent.
- Eigen results are consistently poor
- Table of results can be found here
- Compiler versions
  - G++ 8.3.1
  - Icc 19.1.1.217
  - Pgc++ 19.4-0
  - Nvcc 10.1.168



Backup

Some information about the different implementations used. Flags used in the makefile, division of track bunches, and unique features of each mode

OpenMP

- Make
  - Compilers:
    - Gcc: -O3 -I. –fopenmp –lm -lgomp
    - Icc: -Wall -I. -O3 -fopenmp -march=native -xHost -qopt-zmm-usage=high
    - Pgi: -I. -Minfo=mp -fast -mp -Mnouniform -mcmodel=medium -Mlarge_arrays
  - Version 3
- Set up
  - Nevts 100
  - Nb 600
  - Bsize 16
- Specifics:
  - #pragma omp simd
    - Used in loop over bunches of tracks
    - Used in propagation function
    - Used in unrolled matrix multiplication in error propagation
  - #pragma omp parallel for
    - Used in lopes over events

OpenACC

- Make
  - Compilers
    - pgi: -I. -Minfo=acc -fast -Mfprelaxed -acc -ta=tesla -mcmodel=medium -Mlarge_arrays
  - Version 1 (v5 had a longer memory transfer time. Unclear which version to use)
- Set up
  - Nevts 100
  - Nb 300
  - bsize 32
- Specifics:
  - #pragma acc enter data create(trk[0:nevts*nb], hit[0:nevts*nb], outtrk[0:nevts*nb]) after setup
  - #pragma acc update device(trk[0:nevts*nb], hit[0:nevts*nb]) before iteration loop
  - #pragma acc parallel loop gang collapse(2) present(trk, hit, outtrk) over event loop (and bunches of tracks)
  - #pragma acc routine vector nohost above propagate and error prop functions
  - #pragma acc loop vector in propagate function loop and unrolled matrix multiplication in error propagation
  - #pragma acc update host(outtrk[0:nevts*nb]) after iteration loop

TBB

- Make
  - Compilers:
    - Gcc: -fopenmp –O3 –I. –lm -lgomp
    - Icc: -Wall -I. -O3 -fopenmp -march=native -xHost -qopt-zmm-usage=high
- Set up
  - Nevts 100
  - Nb 600
  - bsize 16
- Specifics:
  - #pragma omp parallel for -> tbb::parallel_for(blocked_range<size_t>…)
- Todo:
  - Check block sizes and any other factors that might affect the overhead time

CUDA

- Make
  - Compilers:
    - Nvcc: -arch=sm_70 --default-stream per-thread -O3 --expt-relaxed-constexpr -I/mnt/data1/dsr/mkfit-hackathon/eigen -I/mnt/data1/dsr/cub
  - Version 1 – unified memory. Version 2 – explicit memory transfers.
- Set up
  - Nevts 100
  - Nb 300
  - bsize 32
  - Grid(10,1,1), block(32,32,1), streams 10
- Specifics:
  - Unified memory uses prefetching and MemAdvise (unclear if memadvise contributes to any speedup).
  - Blocks run over event loop. Theads-y run over bunches of tracks. Threads-x run over propagate function loop and unrolled matrix multiplication in error propagation.
- Todo:
  - Check the memory transfer to ensure data isn't being transferred multiple times because of the streams

Eigen

- Make
  - Compilers:
    - Icc: -fopenmp -O3 -fopenmp-simd -I/mnt/data1/dsr/mkfit-hackathon/eigen -I/mnt/data1/dsr/cub -mtune=native -march=native -xHost -qopt-zmm-usage=high
    - Nvcc: -arch=sm_70 --default-stream per-thread -O3 --expt-relaxed-constexpr -I/mnt/data1/dsr/mkfit-hackathon/eigen -I/mnt/data1/dsr/cub
    - Gcc: -fopenmp -O3 -fopenmp-simd -I/mnt/data1/dsr/mkfit-hackathon/eigen -I/mnt/data1/dsr/cub -lm -lgomp
- Set up
  - Nevts 100
  - Nb 9600
  - bsize 1
- Specifics:
  - All Matrices and vectors are changed to eigen matrix format
  - OpenMP
    - Uses same omp pragmas as openMP version (excluding error prop functions).
    - Error propagation matrix multiplication is done using eigen matrix multiplication (not unrolled explicit multipliction)
  - Cuda
    - Grid(10,1,1), block(32,32,1), streams 10.
    - Division or work same as cuda version but errop propagation matrix multiplication is done using eigen matrix multiplication operator.
- Todo:
  - Redo error propagation and the struct formats to properly use the eigen matrix format (currently the eigen syntax is just jammed in place from the typical way the code runs for other versions).
  - Make another version using the tensor module
- Notes:
  - This is the only version that completely changes the syntax and structure of the data and as a result this seemed to me to be the most difficult to use. The only Eigen specific feature seems to be the matrix multiplication does in the error propagation and it is not clear that it does this better than unrolling the matrices as in the typical omp version.
  - Documentation

Alpaka

- Make
  - Compilers:
    - gcc (icc?): -fopenmp -O3 –I. –lm -lgomp
    - nvcc (pgi?)
  - Other options:
    - -I/mnt/data1/mgr85/p2z-tests/include -DALPAKA_ACC_CPU_BT_OMP4_ENABLED -DALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED -DALPAKA_ACC_CPU_B_SEQ_T_FIBERS_ENABLED -DALPAKA_ACC_CPU_B_OMP2_T_SEQ_ENABLED -DALPAKA_ACC_CPU_B_SEQ_T_OMP2_ENABLED -DALPAKA_ACC_CPU_B_SEQ_T_THREADS_ENABLED
    - -I${TBB_PREFIX}/include -L${TBB_PREFIX}/lib -Wl,-rpath,${TBB_PREFIX}/lib -ltbb -DALPAKA_ACC_CPU_B_TBB_T_SEQ_ENABLED
- Set up
  - Nevts 100, Nb 16, bsize 600
- Specifics:
  - Use alpaka syntax to setup type of accelerator and division of work. Matrices and structs remain the same.
  - Accelerator types
    - AccCpuSerial
    - AccCpuOmp4
    - AccCpuThreads
    - AccCpuOmp2Threads (1 block per grid, threadsperblock(1,16,8)
    - AccCpuOmp2Blocks
    - AccCpuTbbBlocks
    - AccGpuCudaRt
  - Loop over events is made into a kernel for alpaca
    - Thread-z runs over nevts, thread-y runs over bunches of tracks, thread-x runs over propagate function loop and error propagation.

- Todo
  - Get Gpu version running (memory transfers using alpaka syntax) and tbb version
  - Verify the block-thread divisions are optimal for each accelerator type.
    - Attempt vectorization using elements?
  - Include streams
  - Memory transfers regardless of accelerator type?
- Note: most acc types gave poor results scanning along the block-thread divisions. Only AccCpuOmp2Threads gave a result that seemed comparable to other p2z results.

Alpaka Info

- The library is header only and is included using #include <alpaca/alpaca.hpp> and selecting the correct definition for whichever accelerator you want to use.
  - http://alpaka-group.github.io/alpaka/index.html
  - Can be cloned from here: https://github.com/alpaka-group/alpaka
    - Also need to setup boost before compiling: https://www.boost.org/
  - Not many examples for alpaka provided but it wasn't too difficult to get a running version regardless of that.
- Alpaka uses an offloading model separating the host from the accelerator device.
- Method of parallelization is influenced by CUDA. Uses a hierarchy comprised of grids (global mem), blocks (shared mem), threads (register mem) and elements (used for simd vectorization) (in that order).
- Kernel functions are executed by threads
- In theory, different methods or parallelization can be mixed easily.
  - They claim running the application on a new platform requires changing only 1 line, however new block-thread divisions would also have to be set. Memory transfers might also need to be done explicitly as well.
- More information: https://arxiv.org/pdf/1602.08477.pdf

Accelerator Back-end	Lib/API	Devices	Execution strategy grid-blocks	Execution strategy block-threads
Serial	n/a	Host CPU (single core)	sequential	sequential (only 1 thread per block)
OpenMP 2.0+ blocks	OpenMP 2.0+	Host CPU (multi core)	parallel (preemptive multitasking)	sequential (only 1 thread per block)
OpenMP 2.0+ threads	OpenMP 2.0+	Host CPU (multi core)	sequential	parallel (preemptive multitasking)
OpenMP 4.0+ (CPU)	OpenMP 4.0+	Host CPU (multi core)	parallel (undefined)	parallel (preemptive multitasking)
std::thread	std::thread	Host CPU (multi core)	sequential	parallel (preemptive multitasking)
Boost.Fiber	boost::fibers::fiber	Host CPU (single core)	sequential	parallel (cooperative multitasking)
TBB	TBB 2.2+	Host CPU (multi core)	parallel (preemptive multitasking)	sequential (only 1 thread per block)
CUDA	CUDA 9.0-10.2	NVIDIA GPUs	parallel (undefined)	parallel (lock-step within warps)
HIP(nvcc)	HIP 3.1+	NVIDIA GPUs SM 2.0+	parallel (undefined)	parallel (lock-step within warps)