Porting HEP Parametrized Calorimeter Simulation Code to GPUs


May 17 2021
Motivation: Why Parametrized Calorimeter Simulation?

ATLAS needs a very large amount of simulation

- Simulation is paramount for SM and background modeling in most analyses, as well as general detector and upgrade studies
- A significant issue in Run-2 was the lack of MC-based statistics, and will only worsen in Run-3 and beyond without faster production

A very large fraction of the simulation’s computational budget is spent by the LAr Calorimeter

- Parametrized simulation can speed processing up enormously: FastCaloSim

FastCaloSim is small, self contained, few dependencies
CPU Code Profiling

LAr Calorimeter has massive inherent parallelism – many independent cells and associated tasks.

Profiling studies identified likely hotspots that are parallelizable

CUDA kernels created to run these parts on the GPU

- modified data structures
- reimplement Geometry and parametrization tables for GPU – no STL allowed
- 3 kernels:
  - memory initialization
  - main simulation
  - reduction

PERFORMANCE PROFILING

➢ TFCSLateralShapeParametrizationHitChain::simulate() is the most significant routine except I/O (~30%).
➢ TFCSLateralShapeParametrizationHitChain::simulate() The running time scales with the number of events.
➢ TFCSLateralShapeParametrizationHitChain::simulate() is our target to parallelize/port to GPUs.

I/O routines

Timing for 1000 events

Depth 0: hit chains

TFCSLateralShapeParametrizationHit
Chain::simulateTFCSimulationState...
GPU Validation

CUDA has a very good random number generator (cuRAND)

• FCS needs *lots* of random numbers
  • 3 per hit x ~5k hits per event
• much faster than generating on CPU
• but can’t do bitwise comparisons with CPU – only statistical
  • after looking at lots of histograms, results look statistically equivalent

If we sacrifice speed, we can generate random number on CPU, and transfer them to GPU, using these for all calculations on GPU

• compared the results of 62 million hits in the Electron 64 GeV run
• found only 2 hits calculations that ended up in different calorimeter cell
  • slightly different float rounding policies on CPU/GPU
• if we use double precision variables for certain calculations, difference vanishes

Confident that GPU code does the same thing as CPU
CUDA Performance Studies

I/O to read/unpack parametrization files is expensive: ~15s of 30s

Execution only offloaded if >500 hits, otherwise CPU is faster

GPU kernels very short
  • launch latency limited

Better performance if group work between multiple events to give more work to GPU

![FastCaloSim Timing](image-url)
CUDA Performance Studies

I/O to read/unpack parametrization files is expensive: ~15s of 30s

Execution only offloaded if >500 hits, otherwise CPU is faster

GPU kernels very short
  • launch latency limited

Better performance if group work between multiple events to give more work to GPU
CUDA Performance Studies

GPU performs better for higher energy particles (more hits/work)

Grouped work not as effective since regular GPU is already performant

- need to send extra information to GPU when work is grouped between events

---

**FastCaloSim Speedup for Higher Particle Energies**

<table>
<thead>
<tr>
<th>Particle Type</th>
<th>GPU Event Loop</th>
<th>GPUg Event Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>e E=1 TeV/n.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e E=2 TeV/n.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e E=4 TeV/n.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ν E=1 TeV/n.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ν E=2 TeV/n.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ν E=4 TeV/n.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>π E=1 TeV/n.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>π E=2 TeV/n.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Speedup over CPU**
GPU Usage With CUDA

In general, GPU resources are not well used

- kernels are very short, dominated by launch latency overheads
- work size is small, under-utilizing available GPU cores

Can run multiple concurrent process all sharing one (or more) GPUs

- use `nvidia-cuda-mps-server` to share 2 P100s between up to 32 processes
  - device based time slicing of GPU
- curve is mostly flat – nowhere near saturating GPU resources
- can run 62 processes on a V100 w/ 48GB with little impact on performance
Code Portability

Accelerator architectures are proliferating: NVIDIA / AMD / Intel / GPU / FPGA / TPU

We need portable solutions

• don’t have time to rewrite code for each architecture

► Kokkos
► SYCL
► Alpaka
► OpenMP/ACC
► future C++ standards

• each has strengths and weaknesses

<table>
<thead>
<tr>
<th>Accelerators</th>
<th>Intel</th>
<th>NVidia</th>
<th>AMD</th>
<th>FPGA</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel</td>
<td>Aurora</td>
<td>Cori</td>
<td>Piz Daint</td>
<td>Tsukuba</td>
<td>MareNostrum</td>
</tr>
<tr>
<td>AMD</td>
<td></td>
<td>Perlmutter</td>
<td>Frontier</td>
<td>EI Capitan</td>
<td>LUMI</td>
</tr>
<tr>
<td>IBM</td>
<td>Summit</td>
<td>MareNostrum</td>
<td></td>
<td>Sierra</td>
<td></td>
</tr>
<tr>
<td>Arm</td>
<td>Wombat</td>
<td>Alps</td>
<td></td>
<td></td>
<td>Astra*</td>
</tr>
<tr>
<td>Fujitsu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fugaku</td>
</tr>
</tbody>
</table>

- Amazon EC2 P3
- Amazon Graviton2
- Google Cloud TPU
- Microsoft Azure
- Intel DevCloud
HEP-CCE

Three-year (2020-2023) pilot project
  • four US labs, six experiments, ~12 FTE, ~30 collaborators

1. Portable parallelization strategies
   • exploit massive concurrency
   • one Algorithm compiled and run on many architectures
   • use representative HEP testbeds to evaluate technologies

2. Fine-grained I/O and related storage issues
   • new data models (zero-copying, SOA,...)
   • event batching (CPU offloading)

3. Optimizing event generators

4. Running complex workflows on HPCs
   • main initial use case: cosmology surveys

Open collaboration
  • IRIS-HEP, CERN/HSF, UK/SWIFT, SciDAC/RAPIDS
Kokkos
Kokkos provides portability via various backends: OpenMP, CUDA, (tbb), etc
Single source C++, standard dependent on backend (nvcc limits to C++14)
Abstractions usually provided via C++ header libraries
  • parallel_for, reduction, scans
Data interaction via \texttt{Kokkos::Views<...>}
  • explicit memory movement
  • memory space selected via policy
Execution is bound to an Execution Space
  • device backend must be selected at compile time
  • host serial, host parallel, device parallel
Porting FastCaloSim to Kokkos

Build infrastructure

- Kokkos has good CMake integration
- requires separate binaries for each device backend (CUDA, HIP, Intel) or host parallel (pThread, OpenMP)
- In theory you can run both device/host parallel backends in same code, but then you can’t use the default execution space for your kernels: have to say which go where

Shared libraries not compatible with device symbol relocation

- if you want shared libs, all symbols in a kernel must be visible to one compilation unit
  - wrap kernels in one file that does a bunch of #include
  - needed to do some function/file refactoring to make it all work

CUDA backend interoperable with pure CUDA

- can call CUDA functions from Kokkos kernels
- makes incremental porting and validation much easier

All offloaded data structures need to be converted to Kokkos Views
Kokkos: Porting Data Structures

Kokkos Views can either allocate host/device memory, or wrap existing pointers
  • makes incremental porting of cuMalloc memory easier

Supports both row and column major ordering

Jagged multidimensional arrays not well supported by Kokkos Views
  • Views of Views not meant for this
  • lots of extra boilerplate needed to make work
  • easier to flatten to 1D array, or pad to 2D

Requires explicit Host ↔ Device memory migration
  • need to create Views on host to hold copied information

Non-zero overhead to using Views
  • both in the extra steps for creating the host/device Views, and operations on them
  • default memory initialization can be expensive
Kokkos: Porting Kernel Code

While syntax is different from CUDA, concepts are the same

• functions → lambdas
• parallel_for, parallel_scan, reductions
  • some CUDA features not yet available in Kokkos
• atomics (but not between devices or host/device parallel execution spaces)

Most FCS functions identical between CUDA / Kokkos

• use a single file with #ifdef to select attributes to share as much code between version

```cpp
#ifdef USE_KOKKOS
  #include <Kokkos_Core.hpp>
  #include <Kokkos_Random.hpp>
  #define __DEVICE__ KOKKOS_INLINE_FUNCTION
#else
  #define __DEVICE__ __device__
#endif
```
Kokkos: Performance

Exercise various backends, compare to original CUDA

• CUDA reference is NVidia 2080
• HIP on AMD GPU (Vega56)
• pThread / OpenMP best performance with ~15 threads/procs
• HIP\(^2\) is a pure HIP port, run on AMD GPU

Kokkos does not handle GPU memory initialization efficiently
Kokkos kernel launch penalties worse than CUDA

AMD Vega56 has large launch latencies

HIP/AMD uses the CPU a lot more than CUDA when executing kernels on GPU
OpenMP/pThreads only useful if other CPUs unused
Code was ported from CUDA, not rewritten

<table>
<thead>
<tr>
<th>CPU Freq</th>
<th>CUDA</th>
<th>HIP</th>
<th>HIP(^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2200 MHz</td>
<td>5.6</td>
<td>9.4</td>
<td>152</td>
</tr>
<tr>
<td>3700 MHz</td>
<td>3.4</td>
<td>5.3</td>
<td>60</td>
</tr>
</tbody>
</table>

kernel launch latencies / µs
SYCL / dpc++

SYCL 2020 plus Intel extensions

Single source C++ (understands C++17)

Explicit memory transfers not needed
  • builds a DAG of kernel/data dependencies, transfers data when needed on device

Can use Unified Shared Memory

Executes on most architectures
  • including CPU, GPU, FPGA
  • selectable at runtime (with caveats)
  • complex ecosystem

Seen enormous evolution over the past year

Intel wants to push into llvm main branch
  • OneAPI to become an open specification
SYCL Runtime Backends

With much effort, we were eventually able to run FastCaloSim/SYCL on all 3 flavours of GPUs:

- Intel: iGPU, XeLP, XeHP using Intel dpcpp
- NVidia: sm60 – sm80 architectures using llvm/sycl w/ CUDA backend
- AMD: vega56 and MI50 using hipSYCL

Compilers were changing rapidly over the course of the study
- many bugs fixed and new features made available
- not all features available on all platforms
- performance varied wildly as some silicon (eg Intel) is still very alpha/beta

We will present full results at a separate venue
Host Performance of Portability Layers

“Single Source” is a significant selling point for Kokkos and SYCL

- same source code to run on CPU or GPU

At what price?

- Kokkos “serial” performance is 35% worse than native CPU code
- SYCL is somewhat better, but still inferior to native CPU

Kokkos has the ability to do “host parallel”
execution on CPUs

- openMP or pThreads backends
- best performance: 12 threads/processes is only 2x faster than single core native CPU
  - only useful if those other cores have no useful work available

Is single source performant portability a pipe dream?
Lessons Learned

Build configuration requirements may be challenging

- Kokkos shared libs vs relocatable device code: code reorganization
- dpcpp changing rapidly, things that worked last week may not work today (has stabilized recently)

Separate binaries for different device backends

- Kokkos explicitly, SYCL because you need different compiler flavours
- implications for production code distribution

CUDA→ Portability Layer concepts translate well

- Views / Buffers come with overhead / penalties

Launch latencies for tiny kernels kills performance on all platforms

- Portability layers make it worse
- AMD is currently much worse than NVIDIA. Will RDNA2 / CDNA2 / Instinct and better drivers improve things?

High performance single source CPU/GPU may be a pipe dream

- CPU version of Kokkos/SYCL code is significantly slower than native CPU code

GPU very underutilized in FastCaloSim

- grouping data between events helps: may require significant refactoring of frameworks
- a single GPU can be shared between multiple processes
What Comes Next for FastCaloSim

Other Parallel Portability Layers:
- OpenMP / OpenACC
- Alpaka
- evolving C++ standards?

Other backends
- Intel discrete GPU (Arctic Sound/XeHP and Ponte Vecchio/XeHPC via Kokkos and SYCL)
  - we can already run FCS/SYCL on XeLP, XeHP nodes at Argonne, but can’t show ;-(
  - AMD RDNA2 / CDNA2 (MI100)

Investigate poor Kokkos/SYCL serial CPU performance

Better understanding/evaluation/reporting of metrics
- in coordination with other HEP/CCE-PPS testbeds

Update FastCaloSim to reflect what ATLAS is currently using
- more realistic particle scenarios
- integrate into ATLAS repositories
What Recommendations Would We Make Today?

Are you buying hardware for you trigger farm today?
  • NVIDIA / CUDA

Is short term performance the main metric?
  • NVIDIA / CUDA

Is short term performance important, but not ultimate, and want good portability?
  • Kokkos

Do you want to target mainly Intel and NVIDIA GPU hardware?
  • SYCL

Long term portability on all platforms
  • Kokkos

Big Caveat: Non-NVIDIA software/hardware changing very rapidly: these answers may be different in six months.
fin
Acknowledgments

Thanking all the people who contributed to this project

CUDA port:
  Zhihua Dong (BNL)
  Meifeng Lin (BNL)
  Kwangmin Yu (BNL)

Kokkos port:
  Zhihua Dong (BNL)
  Charles Leggett (LBL)

SYCL port:
  Charles Leggett (LBL)
  Vincent Pascuzzi (LBL)

Physics Validation:
  Doug Benjamin (ANL)
  Heather Gray (LBL)
  Tadej Novak (DESY)