Challenges Facing HEP Computing on Heterogeneous Architectures in the Exascale Era

Charles Leggett

Software and Computing Round Table: Programming for Future Architectures
February 18 2020
10 years ago we saw and predicted a major change in computing architecture driven by CPU design limitations

- smaller, less powerful CPUs
- many more cores per CPU
- less memory per core

In response, we have heavily invested in multi-threaded frameworks to better make use available resources
10 years ago we saw and predicted a major change in computing architecture driven by CPU design limitations
- smaller, less powerful CPUs
- many more cores per CPU
- less memory per core

In response, we have heavily invested in multi-threaded frameworks to better make use available resources

Unfortunately, we will need much more computing power in the not so distant future than we have budgeted for
- though it doesn't look as bad as it did last year due to improved parametrized simulation codes
10 years ago we saw and predicted a major change in computing architecture driven by CPU design limitations
- smaller, less powerful CPUs
- many more cores per CPU
- less memory per core

In response, we have heavily invested in multi-threaded frameworks to better make use available resources

Unfortunately, we will need much more computing power in the not so distant future than we have budgeted for
- CMS has similar projections
10 years ago we saw and predicted a major change in computing architecture driven by CPU design limitations
• smaller, less powerful CPUs
• many more cores per CPU
• less memory per core

In response, we have heavily invested in multi-threaded frameworks to better make use available resources

Unfortunately, we will need much more computing power in the not so distant future than we have budgeted for

Frankly, storage / disk resource needs are even more scary, but that's a different talk
Why Are LHC Computing Needs Increasing?

- As the luminosity of the beam increases, the number of interactions per bunch crossing ($\mu$) increases dramatically
  
  - events become much larger
  - tracking becomes much more difficult
  - track combinatorics begin to dominate in the simulation and reconstruction workflows
How to Address the Computing Shortfall

► Most of HEP computing takes place on the "Grid"
  • distributed federation of dedicated, commodity processors
    » broad range of site and CPU performance
      » tens to thousands of nodes
      » >10 year old CPUs to most recent ones
  • well established OS for ease of job deployment
    » containerization has made this less important
  • located primarily in Europe and USA
How to Address the Computing Shortfall

► Most of HEP computing takes place on the "Grid"
  • distributed federation of dedicated, commodity processors
    • broad range of site and CPU performance
      » tens to thousands of nodes
      » >10 year old CPUs to most recent ones
    • well established OS for ease of job deployment
      » containerization has made this less important
  • located primarily in Europe and USA

► In the past several years, there has been an increased use of HPCs
  • usually in opportunistic mode
  • more challenging than Grid computing due to non-standard hardware, OS, and available system software / libraries
  • containerization has reduced many of these issues
  • this may be the way to address our computing shortfall
In the next generation of supercomputers we see extensive use of accelerator technologies:

- **Oak Ridge: Summit (2018)**
  - 4608 IBM AC922 nodes w/ 2x Power9 CPU
  - 3x NVIDIA Volta V100 + NVLink / CPU

- **LBL: NERSC-9 "Perlmutter" (2020)**
  - AMD EPYC "Milan" x86 only nodes + mixed CPU / "next gen" NVidia GPU

- **Oak Ridge: Frontier (2021)**
  - 1.5 exaflop
  - AMD EPYC CPU + 4x AMD "Instinct" GPU

- **LLNL: Sierra (2018)**
  - 4320 IBM AC922 nodes w/ 2x Power9 CPU
  - 2x NVIDIA Volta V100 + NVLink / CPU

- **Argonne: Aurora A21 (2021)**
  - possibly first exascale HPC
  - Intel Xeon CPU + Intel Xe/gen12 GPU + Optane
In the next generation of supercomputers we see extensive use of accelerator technologies:

- **Oak Ridge: Summit (2018)**
  - 4608 IBM AC922 nodes w/ 2x Power9 CPU
  - 3x NVIDIA Volta V100 + NVLink / CPU

- **LBL: NERSC-9 "Perlmutter" (2020)**
  - AMD EPYC "Milan" x86 only nodes + mixed CPU / "next gen" NVidia GPU

- **Oak Ridge: Frontier (2021)**
  - 1.5 exaflop
  - AMD EPYC CPU + 4x AMD "Instinct" GPU

- **LLNL: Sierra (2018)**
  - 4320 IBM AC922 nodes w/ 2x Power9 CPU
  - 2x NVIDIA Volta V100 + NVLink / CPU

- **Argonne: Aurora A21 (2021)**
  - possibly first exascale HPC
  - Intel Xeon CPU + Intel Xe/gen12 GPU + Optane

- **Tsukuba: Cygnus (2020)**
  - 2x Intel Xeon 6162 + 4x NVidia V100 GPU
  - 2x CPU + 4x GPU + 2x Intel Stratix FPGA

- **Japan: Fugaku (2021)**
  - manycore ARM A64fx (48+2)
  - integrated "SVE" 512 bit GPU-like accelerator

- **Spain: MareNostrum**
  - Xeon 8268 + Power9 + V100 GPU

- **Switzerland: Piz Daint**
  - Xeon E5 2690 + NVidia P100 GPU
In the next generation of supercomputers we see extensive use of accelerator technologies

- Oak Ridge: **Summit** (2018)
  - 4608 IBM AC922 nodes w/ 2x Power9 CPU
  - 3x NVIDIA Volta V100 + NVLink / CPU

- LBL: NERSC-9 "**Perlmutter**" (2020)
  - AMD EPYC "Milan" x86 only nodes + mixed CPU / "next gen" NVidia GPU

- Oak Ridge: **Frontier** (2021)
  - 1.5 exaflop
  - AMD EPYC CPU + 4x AMD "Instinct" GPU

- LLNL: **Sierra** (2018)
  - 4320 IBM AC922 nodes w/ 2x Power9 CPU
  - 2x NVIDIA Volta V100 + NVLink / CPU

- Argonne: **Aurora A21** (2021)
  - possibly first exascale HPC
  - Intel Xeon CPU + Intel Xe/gen12 GPU + Optane

- Tsukuba: **Cygnus** (2020)
  - 2x Intel Xeon 6162 + 4x NVidia V100 GPU
  - 2x CPU + 4x GPU + 2x Intel Stratix FPGA

- Japan: **Fugaku** (2021)
  - manycore ARM A64fx (48+2)
  - integrated "SVE" 512 bit GPU-like accelerator

- Spain: **MareNostrum**
  - Xeon 8268 + Power9 + V100 GPU

- Switzerland: **Piz Daint**
  - Xeon E5 2690 + NVidia P100 GPU
Broadly in the next generation of supercomputers we see extensive use of accelerator technologies.

- **Oak Ridge: Summit (2018)**
  - 4608 IBM AC922 nodes w/ 2x Power9 CPU
  - 3x NVIDIA Volta V100 + NVLink / CPU
- **LBL: NERSC-9 "Perlmutter" (2020)**
  - AMD EPYC "Milan" x86 only nodes + mixed CPU / "next gen" NVidia GPU
- **Oak Ridge: Frontier (2021)**
  - 1.5 exaflop
  - AMD EPYC CPU + 4x AMD "Instinct" GPU
- **Commercial clouds:**
  - Brainwave / Azure FPGA
  - Google Cloud TPU
  - Amazon EC2 P3
- **LLNL: Sierra (2018)**
  - 4320 IBM AC922 nodes w/ 2x Power9 CPU
  - 2x NVIDIA Volta V100 + NVLink / CPU
- **Argonne: Aurora A21 (2021)**
  - possibly first exascale HPC
  - Intel Xeon CPU + Intel Xe/gen12 GPU + Optane
- **Tsukuba: Cygnus (2020)**
  - 2x Intel Xeon 6162 + 4x NVidia V100 GPU
  - 2x CPU + 4x GPU + 2x Intel Stratix FPGA
- **Japan: Fugaku (2021)**
  - manycore ARM A64fx (48+2)
  - integrated "SVE" 512 bit GPU-like accelerator
- **Spain: MareNostrum**
  - Xeon 8268 + Power9 + V100 GPU
- **Switzerland: Piz Daint**
  - Xeon E5 2690 + NVidia P100 GPU

US funding agencies have indicated that we will not be able to get allocations on these HPCs if our code does not make use of accelerator hardware.
Differences Between CPU and GPU

► CPU:
  • small number of very complicated cores
    • branch prediction
    • instruction pipelining
    • prefetching
  • multiple levels of large caches
  • low latency

► GPU:
  • very many (100k+) simple cores
    • much more hardware for low precision ops than dp
  • cores in a block operate in lockstep
    • branch mis-prediction causes stalls for many cores
  • small cache, complex memory hierarchy
  • vectorized memory ops
  • high throughput, high latency
  • low power (per FLOP)
### Differences Between CPU and GPU

#### CPU:
- small number of very complicated cores
  - branch prediction
  - instruction pipelining
  - prefetching
- multiple levels of large caches
- low latency

#### GPU:
- very many (100k+) simple cores
  - much more hardware for low precision ops than dp
- cores in a block operate in lockstep
  - branch mis-prediction causes stalls for many cores
- small cache, complex memory hierarchy
- vectorized memory ops
- high throughput, high latency
- low power (per FLOP)

Driving reason for GPU usage in HPCs:
Energy (power and cooling) requirements to deploy a top 10 HPC with traditional architecture are cost prohibitive.
**NVidia V100**

- 6 Graphics Processor Cluster
- 42 Texture Processor Cluster
- 84 Streaming Multiprocessor
- 4x 8 FP64
- 4x 16 FP32
- 4x 16 INT32
- 2 Tensor Core

- 7.8 TFLOP FP64
- 15.7 TFLOP FP32
- 125 TFLOP Tensor matrix mult
- 300 W
**NVidia V100**
- 6 Graphics Processor Cluster
- 42 Texture Processor Cluster
- 84 Streaming Multiprocessor
- 4x 8 FP64
- 4x 16 FP32
- 4x 16 INT32
- 2 Tensor Core

**Modern CPU**
- 1-2 TFLOP FP32
- ~150 W

Modern-ish GPU

- 7.8 TFLOP FP64
- 15.7 TFLOP FP32
- 125 TFLOP Tensor matrix mult
- 300 W
Why GPU Programming Is Challenging

► Separate "kernels" must be written to execute code on GPU
  • using languages like CUDA (NVidia), SyCL (Intel), hip (AMD), OpenACC, Kokkos, …
► In order to take advantage of very wide GPU architectures, need LOTS of available parallelism ( > 100,000)
► GPU threads in a block need to execute same instruction to work efficiently
  • branches and branch misprediction will cause poor GPU performance
► Complex memory hierarchy requires efficient management
  • badly designed interaction between threads and memory locations can kill performance
  • can't allocate new memory on GPU from a kernel: no STL
► Data structures need to be moved to and from GPU
  • large latencies, slow-ish transfer speeds (PCIe, NVLink)
  • conversion overhead if not in GPU-friendly format
► Amdahl's law: our code is made of many, many components
► Validation: different code paths for CPU / GPU
► Debugging is much more challenging
HEP Reconstruction Workflows

- Workflows tend to be composed of many individual tasks, often in a very serial fashion
  - limited inherent concurrency
  - Amdahl's law limits gains if few modules offloaded
HEP Reconstruction Workflows

- Workflows tend to be composed of many individual tasks, often in a very serial fashion
  - limited inherent concurrency
  - Amdahl's law limits gains if few modules offloaded

In general, HEP codes are poorly suited for GPUs:
- many discrete units, often needing serial invocation
  - would need to touch many 100k lines of code
- large amounts of data manipulation and conversion
  - GPU ↔ CPU data transmission latencies are prohibitive
- code structures are very branchy - lots of different paths that are cut dependent
  - GPU threads will stall
- loops are at best a few thousand units wide
  - GPUs work best with >100k units
Non-GPU Accelerators

- GPUs are not the only accelerator on the market
  - FPGA
    - Xilinx
    - Intel Arria-10 (CPU + FPGA)
    - Microsoft Azure / Brainwave (for NN inference)
  - ASIC
    - Intel Nervana for AI (separate chips for training and inference)
    - Google TPU (optimized for TensorFlow)

- Programming for these is much more challenging than for GPUs
  - Intel OneAPI / SyCL claims to target all Intel hardware with same source code

- No large HPC has yet decided to use non-GPU accelerators
  - there are several smaller ones
  - would not be surprised to see CPU + GPU + FPGA in next round
Accelerator Usage In LHC Experiments

▶ **LHCb:**
  - full online HLT1 re-written in CUDA to run on GPUs
  - end-to-end solution, to minimized host ↔ device data transfers
  - still not sure if will implement for Run 3:
    - cost: what do GPUs do when not taking data? (HLT farms are very powerful compute resources)
    - data buses / IO in each Event Builder node already saturated. Adding GPUs may be too much. also heat + airflow issues

▶ **CMS**
  - reconstruction framework (cmssw) supports transparent offloading of modules to accelerator. modules re-activated when kernel has finished, and data is ready
  - ability to do offline tracking (Patatrack)
  - full Pixel, HCAL and ECAL online reconstruction

▶ **Alice**
  - tracking: Full TPC and part of ITS on GPU. Hope to extend to full barrel tracking on GPU
  - extensive memory management via custom allocators on GPU to reuse memory

▶ **ATLAS**
  - evaluated use of GPUs in High Level Trigger for Run 2/3, but decided against it due to cost and **data conversion inefficiencies**. Re-evaluating for Run 4
Accelerator Usage In LHC Experiments

▶ LHCb:
- full online HLT1 re-written in CUDA to run on GPUs
- end-to-end solution, to minimized host ↔ device data transfers
- still not sure if will implement for Run 3:
  - cost: what do GPUs do when not taking data? (HLT farms are very powerful compute resources)
  - data buses / IO in each Event Builder node already saturated. Adding GPUs may be too much. also heat + airflow issues

▶ CMS
- reconstruction framework (cmssw) supports transparent offloading of modules to accelerator. modules re-activated when kernel has finished, and data is ready
- ability to do offline tracking (Patatrack)
- full Pixel, HCAL and ECAL online reconstruction

▶ Alice
- tracking: Full TPC and part of ITS on GPU. Hope to extend to full barrel tracking on GPU
- extensive memory management via custom allocators on GPU to reuse memory

▶ ATLAS
- evaluated use of GPUs in High Level Trigger for Run 2/3, but decided against it due to cost and data conversion inefficiencies. Re-evaluating for Run 4

Experiments have had the best gains using accelerators in the online environment:
- hardware is more stable, and explicitly configured for desired purpose
- tasks are simpler, code less complex
- data structures are often smaller than in offline
- can keep significant fraction if not the entire workflow on accelerator to minimize data transfer penalties
Portability Challenges

- Next 3 major DOE HPCs each use a different GPU manufacturer
  - **Perlmutter**: NVidia
  - **Aurora**: Intel
  - **Frontier**: AMD

- Each manufacturer has a preferred/supported language
  - NVidia: **CUDA**
  - Intel: **dpc++** (OneAPI)
  - AMD: **hip**

- There also exist higher level abstraction layers that hide the specifics of the hardware
  - OpenMP / OpenACC
  - Kokkos / Raja / Alpaka

- Language extensions and application libraries
  - Thrust (stl-like libraries for GPUs)

- There is currently **NO** software solution that allows the same code to run on all three
  - except sort-of OpenMP, which is very non-optimal, and requires a lot of hand tweaking

- LHC experiments have a very long timetable: project to run to 2038 and beyond
  - between them, there are 10s of millions of lines of mostly C++ code
  - can only afford to rewrite **ONCE** to code for accelerators if there's a demonstrated benefit
    - ATLAS took > 3 years to recode for MT safety (and still isn't done)
The software / hardware mapping is somewhat complex. And currently fluid.
Intel has recently further complicated / simplified the situation by announcing it will drop OpenCL.
If you start with a single known hardware architecture, things are a little clearer:
SyCL Ecosystem

SYCL source code

(clang dpc++)

(dpc++)

Any CPU

OpenCL + SPIR-V

- Intel CPUs/GPUs
- other SPIR-V devices?

OpenCL + SPIR(-V)

- Intel CPUs/GPUs
- AMD GPUs (depending on driver stack)
- ARM Mali
- Renesas R-Car

ComputeCpp

PTX devices

- NVIDIA GPUs

(triSYCL)

(experimental!)

Any CPU

OpenCL + SPIR-df

(with OpenMP)

- pocl (CPUs, NVIDIA GPUs)
- Xilinx FPGAs

(hipSYCL)

(experimental!)

Any CPU

OpenCL + SPIR-df

(with OpenMP)

- pocl (CPUs, NVIDIA GPUs)
- Xilinx FPGAs

(sycl-gtx)

OpenCL 1.2
- pretty much anything :)
SyCL / dpc++

- Single source
- C++ (understands C++17)
- No explicit memory transfers
  - builds a DAG of kernel/data dependencies, transfers data as needed
- Executes on all platforms
  - to some extent. AMD support is limited (hipSyCL is a project of a PhD student)
  - including CPU, FPGA
  - choosable at runtime (kinda)
- Intel wants to push into llvm main branch
  - become an open standard, and possibly c++ language extension
- OpenCL IR layer will be replaced by OneAPI "LevelZero"
  - OpenCL v1.2 standard was too limiting
- Codeplay has promised a direct NVidia backend via CUDA calls
  - Codeplay already provides a ptx backend for their SyCL compiler
- Concurrent kernels don't work yet, for ANY backend
Kokkos

► Usually included as header files, as opposed to pre-made library
  • hardware backend can only be selected at compile time
  • can target CPUs (tbb, pthreads, OpenMP) as well as GPUs

► Somewhat less flexible than SyCL
  • hard to explicitly dispatch kernels without use of a parallel_for-like construct
  • need to identify back-end at compilation time (compilation time is loooong)
  • no concurrent kernel execution at this time
    • beta version that explicitly uses CUDA streams

► Has important features that aren't in SYCL
  • reduction construct
  • child tasks
  • more performant (especially if you don't know what you're doing)

► Very good support infrastructure

► Support for Intel GPU and AMD in progress
  • promised sometime this year
OpenMP / OpenACC

- Two similar mechanisms for annotating code to direct the compiler to offload bits of code to other devices
  - uses #pragmas

- OpenMP was really developed for MP on HPC
  - very large and complex standard
  - recently extended to target GPUs
  - very prescriptive: need to tell compiler exactly how to unroll loops
    - have to modify pragmas when move to different GPU architecture

- OpenACC developed explicitly for accelerators
  - lets compiler make intelligent decision on how to decompose problems
  - is a standard that describes what compilers should do, not must
    - different compilers interpret should very differently
  - very strong support in Fortran community

- Hardest to read (IMHO)
- Best supported on HPC
What if we had tasks that could be offloaded?

Significant impedance mismatch:
- V100 has 160,000 threads
- Most of our loops / data structures are much, much less wide than that
  - gang data between events to increase GPU workload? major framework redesign.

Scheduling and execution of concurrent kernels on the GPU likely necessary
- some support (eg CUDA streams), but not extensive and has non-insignificant performance drawbacks
  - significantly limits portability solutions
    - this will (is promised) to change this year (for Kokkos)

Synchronous offloading of CUDA kernels has a major CPU penalty on parent thread
- lots of GPU ↔ CPU driver communication: CPU hardware thread cannot be re-tasked for other work, loosing all benefit of latency hiding.
- asynchronous offloading is much more performant, but currently not supported by ATLAS or LHCb framework (it is by CMS)
- NVidia is aware of the issue
Shifting Paradigms

► Are we approaching the problem the wrong way?

► Instead of trying to make our code work in a poorly matched environment, like pounding a square peg into a round hole, can we re-frame the problem?

► Find tasks that are very well suited for accelerators
  • Machine Learning: can problems be reformulated into ML?
    • use Graph NNs for track finding instead of Kalman filters
    • lots of other pattern identification-like tasks exist, such as calorimeter cell clustering
    • hyperparameter searches
  • Event Generation for Simulation
    • madgraph and sherpa should work well on GPUs
  • Apply lessons learned from GeantV vectorization of detector geometries
  • Use RTX cores on NVidia for particle propagation
    • chargless for now, but maybe we can convince NVidia to add curved line mechanics to future RTX cores!
Final Thoughts

- The era of exascale HPCs has brought us to a place that we didn't want to go
  - in pursuit of higher FLOP counts and energy efficiency, we've been forced to embrace an architecture that is very ill-suited for HEP computing (and many other kinds of science too)
  - future architectures may be even more radical

- The accelerator and software tool ecosystem is complex, and rapidly changing
  - we expect major developments in the software stack in the coming year
  - we will probably see greater divergence in "gamer" cards vs "compute" cards
    - RT cores, FP64 units, etc
  - luckily, there seems to be a major push towards open standards adoption from all the major vendors
    - unfortunately these seem to be competing standards
  - the US DOE labs are giving large amounts of money to NVidia/Intel/AMD to develop software tools for accelerators

- It would behoove everybody tremendously if all these efforts were coordinated, and a single unified standard was developed. Otherwise the challenges of performant portability on heterogeneous architectures may prove overwhelming.
How To Use Accelerators In HEP

- Find sufficiently "wide" problems
- Use appropriate (flat) data structures
- Keep data as long as possible on accelerator

- Use existing libraries and tools
  - cuBLAS
  - TensorFlow
  - pyTorch, numba, etc

- Rewrite your existing algorithms to use techniques that work well on accelerators
  - massive parallelism
  - machine learning

- If all else fails:
  - Learn CUDA (or maybe SyCL / Kokkos)
  - Learn about the hardware to understand memory and thread hierarchy
  - Profile your algorithms to find offloading candidates