



A Lattice QCD code to use all US Exascale Computers

**Peter Boyle,
Brookhaven National Laboratory
Edinburgh University**

- **Exascale programming challenge: platform performance portability**
 - **CPU SIMD, OpenMP**
 - **CUDA, HiP, SYCL**
- **Grid port**
 - **How we ported ?**
- **GridBench - performance scouting**
 - **What will be required for performance**

**\$50,000 accelerated node with 4TB/s HBM memory, 80TF/s fp32 “typical”
Cray SHASTA - 400 GB/s interconnect (bidirectional)**

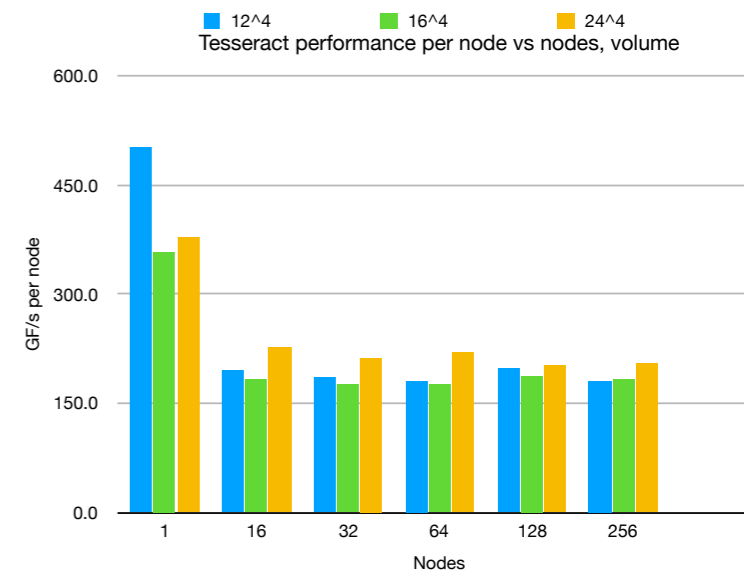


Perlmutter AMD CPU, Nvidia GPU; **CUDA**

Frontier AMD CPU, AMD GPU; **HIP**



Aurora Intel CPU, Intel GPU; **SYCL**



Tesseract, Intel Skylake + OPA; **OpenMP, SIMD**

Grid background:

- Grid is a C++11 high level data parallel interface for cartesian Grid problems
- Principal target is Lattice Quantum Chromodynamics (LQCD).
- Emerged from Intel Parallel Computing Centre funding in 2014, regular iXPUG participant.
 - Grid started after PB conversation with Andrew Richards (CTO of Codeplay) about Pauli matrices, LLVM and games programming
 - *Amusing connection to significant players in SyCL.*
- **Grid aims to give performance portability between many Exascale architectures**
- Grid adopted by USQCD ECP project in 2017/8 as a portability plan of record.
 - PB only non-US investigator on their DOE ECP grant
 - Interfaced to CPS and MILC codes
 - Used in UK, Japan, Germany, CERN, USA
- Multicore performance and portability has been good for some time
- **CUDA GPU support has been added recently**
- Plan to support ALL DOE Exascale systems
- Discuss generalisation to HIP and SyCL

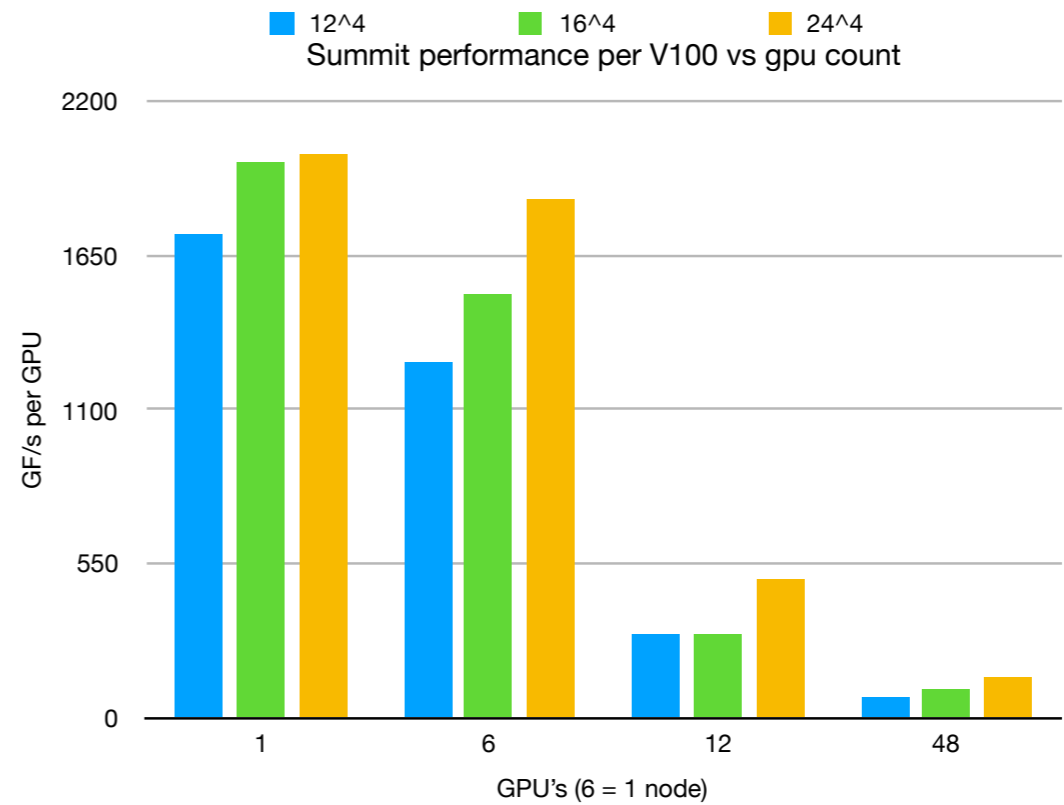
Grid single node performance

Architecture	Cores	GF/s (Ls x Dw)	peak
Intel Knight's Landing 7250	68	770	6100
Intel Knight's Corner	60	270	2400
Intel Skylakex2	48	1200	9200
Intel Broadwellx2	36	800	2700
Intel Haswellx2	32	640	2400
Intel Ivybridgex2	24	270	920
AMD EPYCx2	64	590	3276
AMD Interlagosx4	32 (16)	80	628
Nvidia Volta	84 SMs	1500	15700

- Dropped to inline assembly for key kernel in KNL and BlueGene/Q
- EPYC is MCM; ran 4 MPI ranks per socket, one rank per die
- Also: ARM Neon and ARM SVE port

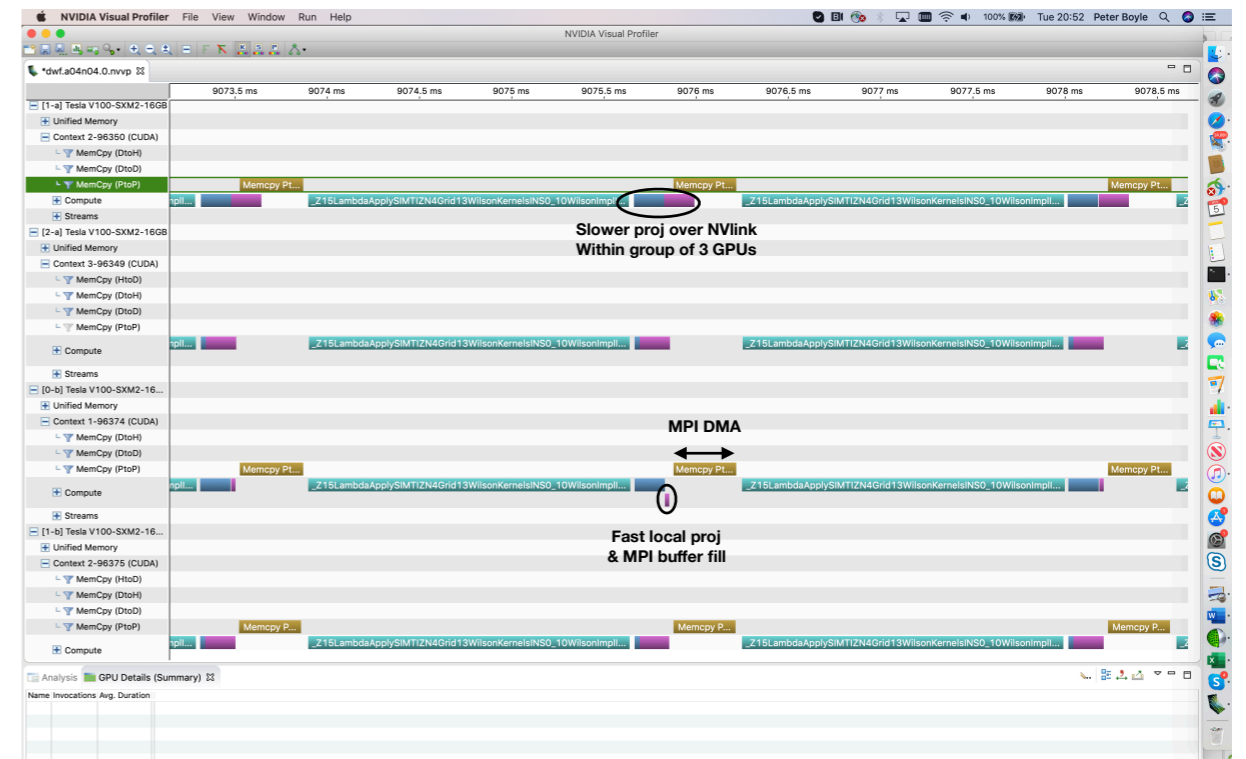
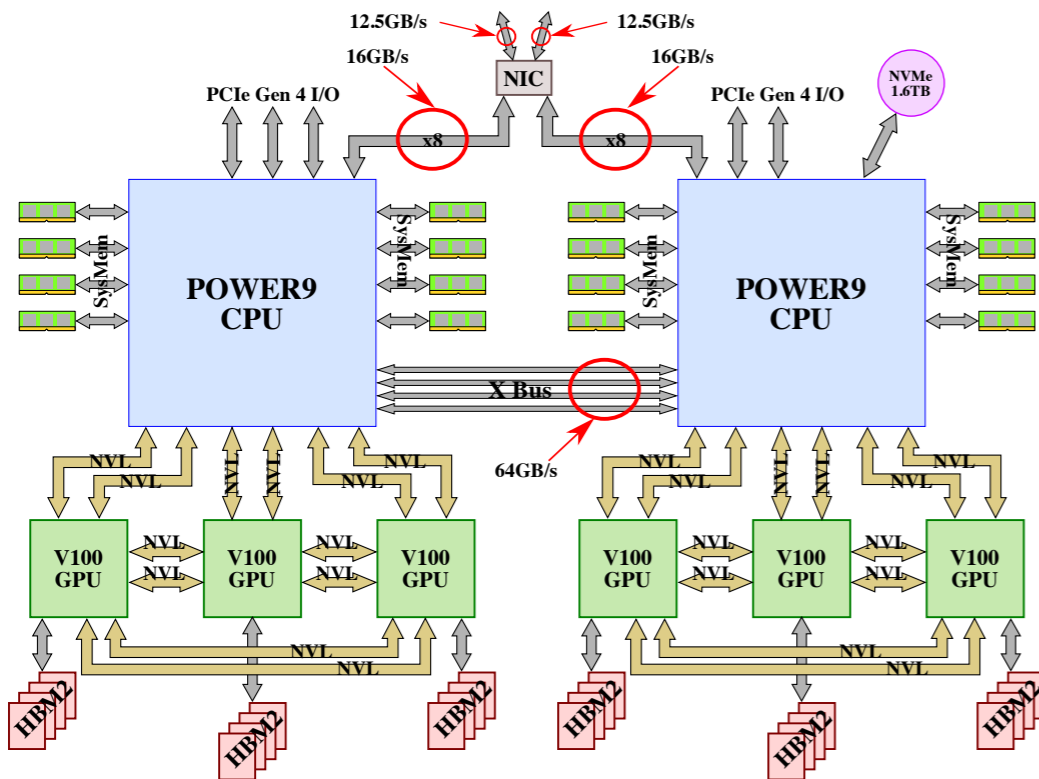
Common source *accelerator port.*

- Assumed Unified Virtual Memory (not restriction to Nvidia as Intel and AMD GPU's support under OpenCL/Linux)
- CUDA; considering OpenMP 5.0 and SyCL for AMD & Intel accelerator portability



Long term resolution:

- Peer-2-peer memory over NVlink, MPI over X Bus.
- 6.5TF/s
- 7TF/s on 144x24x24x24 !!!



Original multicore code:

- Implement vComplex, vReal fundamental vector types in inline intrinsics on CPUs (+ additional compiler vectorisation targets).

SSE, AVX, AVX2, AVX512, BGQ, ARM Neon, ARM SVE ports

CUA GPU extension:

- Neat programming approach taken to write single kernels that give high performance on *both* SIMD and SIMT architectures

Grid QCD code

Design considerations

- Performance portable across multi and many core CPU's

$\text{SIMD} \otimes \text{OpenMP} \otimes \text{MPI}$

- Performance portable to GPU's

$\text{SIMT} \otimes \text{offload} \otimes \text{MPI}$

- N-dimensional cartesian arrays
- Multiple grids
- Data parallel C++ layer : Connection Machine inspired

- **Portability 101:**

- *Always* abstract vendor specific interfaces & types as internal interface in code

- CUDA,
- x86 intrinsics
- MPI

- Reimplement the abstracted interface according to compile target

- **Performance Portability 201:**

- *Always* abstract data layout as vector length, cacheline size etc... may influence

Grid internal interface to acceleration

- OpenMP, CUDA, HIP, SyCL versions

```
// Function attributes
//     accelerator
//     accelerator_inline
// Parallel looping
//     accelerator_for(iter1, num1, nsimd, ... )
//     accelerator_for2d(iter1, num1, iter2, num2, nsimd, ... )
//     accelerator_forNB, accelerator_for2dNB
//     uint32_t accelerator_barrier();           // device synchronise
//
// Parallelism control: Number of threads in thread block is acceleratorThreads*Nsimd
//     acceleratorInit();
//     uint32_t acceleratorThreads(void);
//     void     acceleratorThreads(uint32_t);
//     void     acceleratorSynchronise(void);    // synch warp etc..
//
//     int      acceleratorSIMTlane(int Nsimd); // my location
//     coalescedRead/coalescedReadPermute/coalescedWrite // Memory representation to stack
//
// Reduction
//     template<class t> accelerator_sum(t *tp,uint64_t num)
//
// Memory management:
//     void *acceleratorAllocShared(size_t bytes);
//     void *acceleratorAllocDevice(size_t bytes);
//     void acceleratorFreeShared(void *ptr);
//     void acceleratorFreeDevice(void *ptr);
//     void *acceleratorCopyToDevice(void *from,void *to,size_t bytes);
//     void *acceleratorCopyFromDevice(void *from,void *to,size_t bytes);
```

Capturing SIMD and SIMT in a single code

SIMT and SIMD portability: per thread “stack” computational types *differ*

The struct-of-array (SoA) portability problem:

- Scalar code: CPU needs struct memory accesses struct calculation
- SIMD vectorisation: CPU needs SoA memory accesses and SoA calculation
- SIMT coalesced reading: GPU needs SoA memory accesses struct calculation
- GPU data structures in memory and data structures in thread local calculations *differ*

Model	Memory	Thread
Scalar	Complex Spinor[4][3]	Complex Spinor[4][3]
SIMD	Complex Spinor[4][3][N]	Complex Spinor[4][3][N]
SIMT	Complex Spinor[4][3][N]	Complex Spinor[4][3]
Hybrid?	Complex Spinor[4][3][Nm][Nt]	Complex Spinor[4][3][Nt]

How to program portably?

- Use operator() to transform memory layout to per-thread layout.
- Two ways to access for read
- operator[] returns whole vector
 - operator() returns SIMD lane threadIdx.y in GPU code
 - operator() is a trivial identity map in CPU code
- Use coalescedWrite to insert thread data in lane threadIdx.y of memory layout.

Grid buys into advanced C++ to abstract architecture dependence - aligns with SYCL

Granularity exposed through ISA/Performance

⇒ data structures must change with each architecture

OpenMP, OpenAcc do not address data layout

Several packages arriving at similar conclusions:

- Kokkos (Sandia)
- RAJA (Livermore)
- Grid (Edinburgh + DOE ECP)

Use advanced C++11 features, **inline header template library**, auto, decltype etc..

- Discipline and coding standards are required. C++ can be inefficient otherwise
- Hide data layout in opaque container class
- **Device lambda capture** key enabling feature (CUDA, SyCL), or OpenMP 5.0

```
    accelerator_for(iterator, range, {  
        body;  
    });
```

Unify CPU SIMD and GPU SIMT style of coding with C++ type abstraction

WAS

```
LatticeFermionView a,b,c;  
accelerator_for(ss,volume, {  
    a[ss] = b[ss] + c[ss] ;  
});
```

NOW

```
LatticeFermionView a,b,c;  
accelerator_for(ss,volume,Spinor::Nsimd(), {  
    coalescedWrite(a[ss], b(ss) + c(ss) );  
});
```

On GPU accelerator for sets up a volume \times Nsimd thread grid.

- Each thread is responsible for one SIMD lane

On CPU accelerator for sets up a volume OpenMP loop.

- Each thread is responsible for Nsimd() SIMD lanes

Per-thread datatypes inside these loops cannot be hardwired.

C++ auto and decltype use the return type of operator () to work out computation variables in architecture dependent way.

Single source high performance kernels are optimal on BOTH CPU and CUDA

Porting Grid Cuda code to SYCL

- Intel memory model extensions to SyCL made this easier
 - First pass port was several months (G. Filaci, paid by Intel in Edinburgh)
 - After waiting for these, due to internal abstraction, porting was a few days
- Buffer semantics are too far from other environments for a pleasant, maintainable and portable unified code. Intel's updates to spec v. important.
- Prefer to store and access pointers

```
// CUDA specific
accelerator_inline int acceleratorSIMTlane(int Nsimd) {
    return threadIdx.z;
}
#define accelerator_for2dNB( iter1, num1, iter2, num2, nsimd, ... )\
{ \
    typedef uint64_t Iterator; \
    auto lambda = [=] accelerator \
        (Iterator iter1,Iterator iter2,Iterator lane) mutable { \
        __VA_ARGS__; \
    }; \
    int nt=acceleratorThreads(); \
    dim3 cu_threads(acceleratorThreads(),1,nsimd); \
    dim3 cu_blocks ((num1+nt-1)/nt,num2,1); \
    LambdaApply<<<cu_blocks,cu_threads>>>(num1,num2,nsimd,lambda); \
}
```

```
// SYCL specific
accelerator_inline int acceleratorSIMTlane(int Nsimd) {
    return __spirv::initLocalInvocationId<3, cl::sycl::id<3>>()[2];
}

#define accelerator_for2dNB( iter1, num1, iter2, num2, nsimd, ... ) \
    theGridAccelerator->submit([&(cl::sycl::handler &cgh) { \
        unsigned long nt=acceleratorThreads(); \
        unsigned long unum1 = num1; \
        unsigned long unum2 = num2; \
        cl::sycl::range<3> local {nt,1,nsimd}; \
        cl::sycl::range<3> global{unum1,unum2,nsimd}; \
        cgh.parallel_for<class dslash>( \
            cl::sycl::nd_range<3>(global,local), \
            [=] (cl::sycl::nd_item<3> item) mutable { \
                auto iter1 = item.get_global_id(0); \
                auto iter2 = item.get_global_id(1); \
                auto lane = item.get_global_id(2); \
                { __VA_ARGS__ }; \
            }); \
    });
```

At same time did HIP interface for AMD GPU

Semantically similar to CUDA

```
// HIP specific
accelerator_inline int acceleratorSIMTlane(int Nsimd) {
    return hipThreadIdx_z;
}
#define accelerator_for2dNB( iter1, num1, iter2, num2, nsimd, ... ) \
{ \
    typedef uint64_t Iterator; \
    auto lambda = [=] accelerator \
        (Iterator iter1,Iterator iter2,Iterator lane ) mutable { \
        { __VA_ARGS__; } \
    }; \
    int nt=acceleratorThreads(); \
    dim3 hip_threads(nt,1,nsimd); \
    dim3 hip_blocks ((num1+nt-1)/nt,num2,1); \
    hipLaunchKernelGGL(LambdaApply,hip_blocks,hip_threads, \
        0,0, \
        num1,num2,nsimd,lambda); \
}
```

https://www.olcf.ornl.gov/wp-content/uploads/2019/10/CAAR_HIP_on_Frontier.pdf

- Grid initially used UVM; found performance issues on Summit
- *Optionally now have software cache on device*
 - *My own cross platform version of Sycl Buffer*
 - Would have been useful for KNL
 - Non-uniform memory distinct from GPU architecture, debug on laptop!
 - Balance of probability: at least one vendor will suck at UVM!

Memory in Device Code

- Threads by default can dereference pinned host memory in device code:
 - Memory allocated by `hipHostMalloc()` (more details later)
 - Data travels over host<->device data fabric (e.g. PCIe®)
 - Access will likely be slow compared to other memory types.
- Threads can all access pointers to Unified Virtual Memory:
 - Memory allocated by `hipMallocManaged()`
 - Memory is automatically migrated between host and device by the HIP runtime
 - Can have significant overhead, even when memory is already resident on device
 - Sometimes useful to use UVM in porting process
 - **Highly recommended to migrate away from UVM usage for performance sensitive regions.**
- Threads can all access device global memory via device pointers:
 - Memory allocated by `hipMalloc()`
 - Access is slow compared to more local memory (registers and LDS)
 - Bandwidth can be significantly improved if the wavefront accesses memory in coalesced fashion (more later)

Home Grown Caching interface

- Introduce **MemoryManager** class.
 - Access is made through **View** objects with **RW** intent & location, open/close semantic.
 - Automatic for data parallel expression template engine

● **A=B*C+Cshift(D,1,Xdir);**

This is a data parallel interface across all nodes, array elements
dpcpp is not data parallel in same F90 sense.

- Under the hood, communication is performed
 - Data is possibly moved to/from device
 - Works on KNL, HIP, CUDA and SYCL. Buffer works only on SYCL.
- **MemoryManager** tracks entire buffers in a *software cache* with **O(1)** overhead
 - **O(1)** Hash table maps Host pointers to cache table entries
 - Consistent, CpuDirty, AccDirty states
 - (I prev. designed the IBM BlueGene/Q multicore L1p prefetching cache... :))
 - Linked list LRU queue prioritises evictable arrays
 - **O(1)** push, pop and erase via indirection from Hash table
 - **EvictNext or EvictLast prio for user avoidance of cache thrashing. Full control of algorithm**

SYCL porting details

Current status

System	Technology	accelerator	works	fast
Perlmutter	Cray/AMD Milan/Nvidia A100(?)	cuda	yes	yes
Frontier	Cray/AMD Epyc/Radeon	hip	yes	maybe
Aurora	Cray/Intel SR/Intel Xe	sycl	yes	no
x86 SIMD	AMD	openmp	yes	yes
x86 SIMD	Intel	openmp	yes	yes
Fujitsu Fugaku	ARM SVE	openmp	yes	no

The Grand Plan (1 year timeframe?)

System	Technology	accelerator	simd	future simd ?
Perlmutter	Cray/AMD Milan/Nvidia A100(?)	cuda	GPU	RRRRRIII ?
Frontier	Cray/AMD Epyc/Radeon	hip	GPU	RRRRRIII ?
Aurora	Cray/Intel SR/Intel Xe	sycl	GEN	RRRRRIII ?
Fugaku	Fujitsu/ARM SVE	openmp	GEN	RRRRRIII
x86 SIMD	AMD Rome/Milan	openmp	AVX2	RRRRRIII ?
x86 SIMD	Intel SapphireRapids	openmp	AVX512	RRRRRIII ?

- Introduced RRRR RRRR IIII IIII compiler vectorised layout

```
typedef float vfloat __attribute__((vector_size (8*sizeof(float))));
template<class datum> struct datum2 {
    datum x;
    datum y;
};
template<class pair> struct Complex {
    pair z;
}
typedef datum2<vfloat>    vfloat2;
```

```
typedef GpuComplex<float2 > GpuComplexF;
```

- Aim for *,*,*,* and +,+,+,+ and -,,-,-,- arithmetic signatures and compiler vectorisation. _builtin_permute ??
- NB doubles register footprint on AVX512
- dpcpp/sycl : **get 160 GF/s single precision on 6 skylake cores for L3 resident.** Just as good as direct AVX2.
- Get 55-85 GF/s single precision on Gen9 (24, 48 EUs).

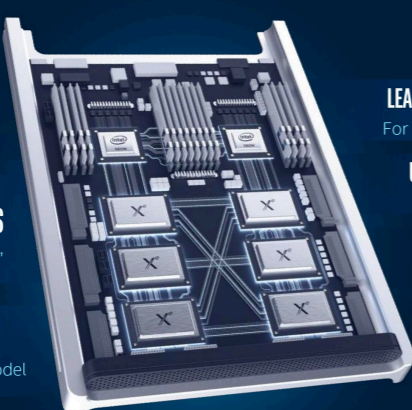
- **Issue #1750**
 - **Multi-GPU memory sharing interface is not defined. Needed for NVlink equivalent intra-node comms.**
 - **Buffer model is weak - consistency between CPU and target device**
 - **Absence of pointers in original model means bouncing through CPU required**
 - **No way to refer to data on two GPUs for intra node transfer!**

Aurora: Bringing It All Together

2 INTEL XEON SCALABLE PROCESSORS
"Sapphire Rapids"

6 X^E ARCHITECTURE BASED GPU'S
"Ponte Vecchio"

ONEAPI
Unified programming model



LEADERSHIP PERFORMANCE
For HPC, data analytics, AI

UNIFIED MEMORY ARCHITECTURE
Across CPU & GPU

ALL-TO-ALL CONNECTIVITY WITHIN NODE
Low latency, high bandwidth

UNPARALLELED I/O SCALABILITY ACROSS NODES
8 fabric endpoints per node, DAOS

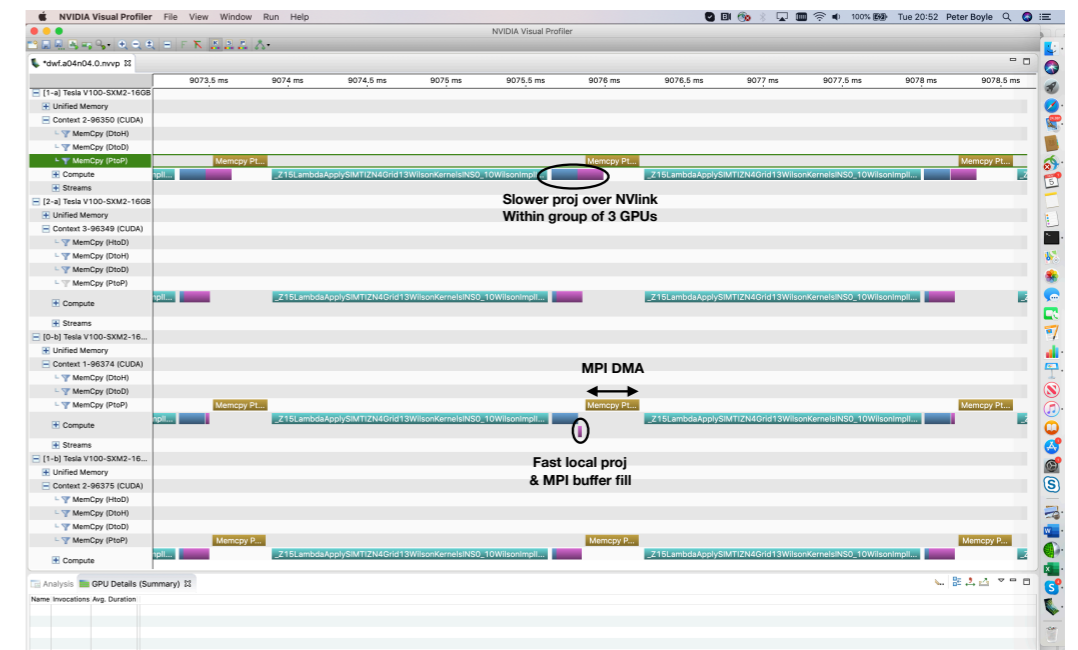
DELIVERED IN 2021

U.S. DEPARTMENT OF ENERGY | Argonne NATIONAL LABORATORY | intel | CRAY

News Under Embargo: November 17, 2019 – 4:00 p.m. Pacific Time

intel | 16

- **Long term resolution:**
 - Peer-2-peer memory over NVlink, MPI over X Bus.
 - **6.5TF/s**
 - **7TF/s on 144x24x24x24 !!!**



Performance scouting with GridBench

- Scout performance optimisation with a heavily cut down code
 - GridBench
- Identified 32bit load per “work item” or “thread” on Gen9
 - Code wants 128bit (complex<double>) vectors with 128bit per work item
 - Nvidia provides 128bit loads with dynamic (hardware detect) coalescing

```
LD.E.128 R4, [R10]
```

	T0	T1	T2	T3
R4	0	4	8	12
R5	1	5	9	13
R6	2	6	10	14
R7	3	7	11	15

Coalesced load = single L1 access

01 = real part

23 = imaginary part

	T0	T1	T2	T3
R4	4	0	12	8
R5	5	1	13	9
R6	6	2	14	10
R7	7	3	15	11

Coalesced load = single L1 access
even if permutation of line

● Summary:

- Grid runs successfully on HIP, SyCL, CUDA
- Nvidia
 - acceleration runs well in production on Summit
- AMD
 - HIP on Nvidia runs well on Summit
 - Waiting for AMD GPU access
- Intel
 - Functional port to SYCL successful and relatively easy
 - Some requests for additions to standard
 - Intel's memory model changes to SYCL were a BIG help
 - pointer additions required for multi-GPU and MPI from device anyway
- Benefitted from already abstracted code to cover
 - Accelerator vs Threading
 - SIMT and SIMD
- Massive computing resources in future
 - Requires investment to exploit successfully
 - Grid is about as ready as it could be given hardware not available yet