Grid: OneCode and FourAPIs

www.github.com/paboyle/Grid

Azusa Yamaguchi *(University of Edinburgh)*
Peter Boyle (*Brookhaven National Laboratory)*

Background:

Grid is a C++11 high level library for lattice Gauge theory[1,2]

It aims to be performance portable across all modern architectures

A number of LQCD software efforts make use of Grid for actions, algorithms (solvers, multigrid, HMC, contraction primitives) [3].

Portability in across the Exascale roadmap requires support for AMD (HIP), Intel (SYCL) and Nvidia (QUDA) GPUs in addition to vectorising multicore architectures. Grid is a single source and targets all of these APIs portably.

Frontier AMD CPU, AMD GPU; HIP

Perlmutter AMD CPU, Nvidia GPU; CUDA

Aurora Intel CPU, Intel GPU; SYCL

Summit IBM CPU, Nvidia GPU; CUDA

+ CPU computing is also *not* going away

Covariant programming:

SIMD and SIMT differ semantically in whether local variables are vectors or scalars

Naively poses a barrier to writing single source kernels which vectorise on CPUs and read coalesce on GPUs.

C++ automatic type inference lets you avoid naming the types (vector or scalar) so you can deduce the type according to the architecture following simply programming idioms.

Combined with *accelerator_for* abstraction capturing and offloading loop bodies in *device lambda functions* we can write high performance kernel code that runs on all four API's.

FourAPIs and OneCode

Portability 101 – abstract the interfaces

```
// CUDA specific
accelerator_inline int acceleratorSIMTlane(int Nsimd) {
 return threadIdx.x;
}
#define accelerator_for2d( 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); \
 }
```

Performance Portability 102 – abstract the layout

```
// SYCL specific
accelerator_inline int acceleratorSIMTlane(int Nsimd) {
return __spirv::initLocalInvocationId<3, cl::sycl::id<3>>()[2];
}
#define accelerator_for2d( 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__ };                                                \
  });                                                              \
 });
```

Similar ideas to RAJA and Kokkos – use device lambda capture ;
lean internal interface to offload - HIP and OpenMP similar

HIP and OpenMP targets

```
// HIP specific
accelerator_inline int acceleratorSIMTlane(int Nsimd) {
 return hipThreadIdx_z;
}
#define accelerator_for2d( 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);                         \
  }
```

```
// OpenMP specific
#define accelerator
#define accelerator_inline strong_inline

#define accelerator_for(iterator,num,nsimd, ... )  \
 thread_for(iterator, num, { __VA_ARGS__ });

#define accelerator_for(iterator,num,nsimd, ... ) \
 thread_for(iterator, num, { __VA_ARGS__ });

#define accelerator_barrier(dummy)

#define accelerator_for2d(iter1, num1, iter2, num2, nsimd, ... )\
 thread_for2d(iter1,num1,iter2,num2,{ __VA_ARGS__ });
```

Wilson dslash kernel (sketch)

Same optimised kernel transforms covariantly between GPU and CPU

Return type of coalescedRead dictates whether SOA or scalar structs are processed in each logical thread

Thread interleaving happens naturally on GPU with memory resident data stored in SOA.

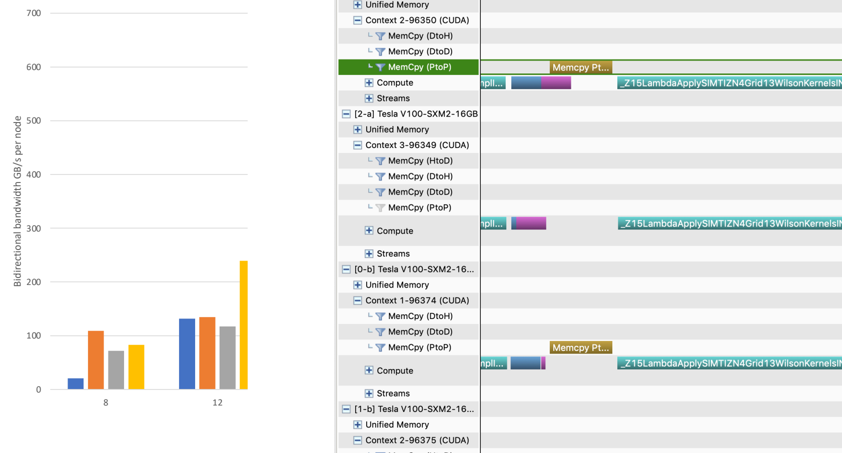
CPU processes loop as SOA data with good vectorisation.

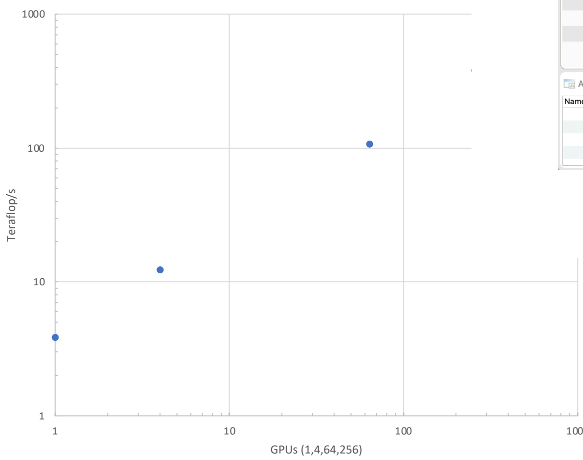
```
template <class Impl> accelerator_inline
void WilsonKernels<Impl>::GenericDhopSite(StencilView &st,
                 DoubledGaugeFieldView &U, SiteHalfSpinor *buf, int sF,
                 int sU, const FermionFieldView &in, FermionFieldView &out)
{
 typedef decltype(coalescedRead(buf[0])) calcHalfSpinor;
 typedef decltype(coalescedRead(in[0]))  calcSpinor;
 calcHalfSpinor chi, Uchi;
 calcSpinor result;
 StencilEntry *SE;
 const int Nsimd = SiteHalfSpinor::Nsimd();
 const int lane=acceleratorSIMTlane(Nsimd);
...
 SE = st.GetEntry(ptype, Dir, sF);
 if (SE->_is_local) {
  int perm= SE->_permute;
  auto tmp = coalescedReadPermute(in[SE->_offset],ptype,perm,lane);
  spProj(chi,tmp);
 } else {
  chi = coalescedRead(buf[SE->_offset],lane);
 }
 acceleratorSynchronise();
 Impl::multLink(Uchi, U[sU], chi, Dir, SE, st);
 Recon(result, Uchi);
...
 coalescedWrite(out[sF],result,lane);
}
```

Nvidia performance:

Excellent performance on ATOS sequ
(e.g. Juelich Booster + Edinburgh Tu
6TF/s per node in multi-node operatic

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





Nsight compute indicates on A100-40

- 82% of L2 cache saturation,
- 76% of HBM saturation
- 36% FMA pipeline usage

Intel / Aurora

- Saturates memory bandwidth on both Iris XE max (DG1) and Arctic Sound
- AMD/Frontier – code ports and run, but performance is a work in progress.
- Fugaku port by Nils Meyer / Regensburg

Device	Fp32 Dw GF/s	Memory BW GB/s
DG1	170-201 GF/s	58 GB/s
V100	1750 GF/s	850GB/s

Conclusions:

Supercomputing companies are not helping scientific productivity with a proliferation of programming models.

There is hidden commonality as they are all based on vector computing

The semantic differences between CPU and GPU can be abstracted and high performance portable source written

Other codes may wish to adopt these techniques

1. Grid: A next generation data parallel C++ QCD library : arXiv:1512.03487
2. Performance Portability Strategies for Grid C++ Expression Templates : arXiv:1710.09409
3. Hadrons: https://github.com/aportelli/Hadrons
   GPT: https://github.com/lehner/gpt
   CPS: https://github.com/RBC-UKQCD/CPS
   MILC: http://www.physics.utah.edu/~detar/milc/