Computational Methods (practice) - Lecture 2

Peter Boyle (BNL, Edinburgh)

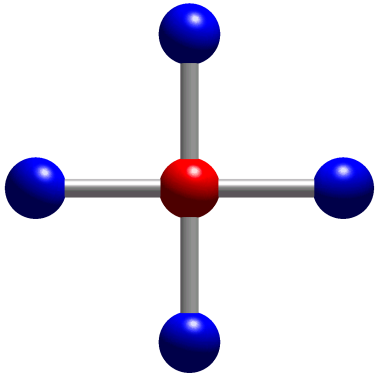
- More on CPUs and GPUs
- Stencil implementation of sparse matrices
- Polynomials of sparse matrices
- Covariant smearing
- Chebyshev polynomials & approximation

Sparse matrices, stencils & PDEs

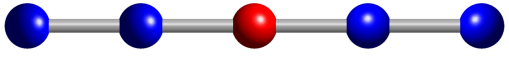
- Encode differential operators as discrete finite difference derivatives
- Think of as sparse matrices - typically band diagonal

$$M(x,y) = \frac{\delta_{x+\mu,y} + \delta_{x-\mu,y} - 2\delta_{x,y}}{a^2}$$

- In typicaly PDE usage, discretisation schemes can be encoded via coefficients and a *stencil*
- The *stencil* is a geometric shape encoding the set of points y that contribute to the operator at point x
- This is translationally invariant and characteristic of a discretisation scheme.
- Naive Laplacian and Wilson Fermions have a $2N_d + 1$ point stencil
- Improved staggered Fermions n discretisations have a $4N_d + 1$ point stencil
- Fourier transformation $\Rightarrow$ phase factor on each term in the stencil defines the lattice Feynman rules
- More non-local actions allow higher order improvement (classical/perturbative/NP)
  c.f. Hamber-Wu, Naik terms.



2D Wilson stencil



Higher order derivative stencil

$$c_1(e^{ip_\mu} - e^{-ip_\mu}) + c_2(e^{i2p_\mu} - e^{-i2p_\mu}) \rightarrow ip_\mu + O(p_\mu^5)$$

Stencils in Grid code

- Grid provides a Stencil object, templated for any field
  - Geometrical object only: decouples data motion (MPI, halo exchange), data access (indexing) from;
  - Algebra on internal indices (multiplying by gauge links, spinor structures, etc)
- Compressor: object knows about Wilson spin projection (2spin/4spin), reduced precision
- User writes a (node local) computational kernel that encodes the physics / internal index manipulation
- Stencil object performs the data motion and presents neighbour tables

What is a CPU execution unit

- One instruction fetch entity: fetches and decodes sequential instuctions until it hits a "branch"
- One *stack pointer* for function data
- (16) Scalar integer registers for address calculations (pointers, 8 bytes)
- (32) Vector floating point registers (e.g. 64 byte wide, or 16 single precision words)
- Vector loads contiguous data, naturally aligned

What is a GPU execution unit

https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html

- Up to 64 instruction fetch entities (warps): fetches and decodes sequential instuctions until it hits a "branch"
- Each instruction fetch controls up to 32 arithmetic units & 32 columns of register file
- Vector integer registers (32 words) for address calculations (pointers)
- 32 *stack pointers* for function data per warp
  these will be "related" to each other if all threads in warp make the same branches and calls
- Vector floating point registers (32 words)
- Vector loads can load unrelated data; contiguous data cases are dynamically detected and optimised in hardware
  Nvidia calls this "coalesced reads/writes"

What is a GPU execution unit

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture

4.1. SIMT Architecture

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. The term *warp* originates from weaving, the first parallel thread technology. A *half-warp* is either the first or second half of a warp. A *quarter-warp* is either the first, second, third, or fourth quarter of a warp.

When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a *warp scheduler* for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. Thread Hierarchy describes how thread IDs relate to thread indices in the block.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp executes each branch path taken, disabling threads that are not on that path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

The SIMT architecture is akin to SIMD (Single Instruction, Multiple Data) vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional code: Cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually.

How do you programme for both at a low level?

Covariant programming : capturing the variation between SIMD and SIMT in a single code

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.

How do you programme for both at a low level?

Covariant programming : capturing the variation between SIMD and SIMT in a single code

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

How do you programme for multiple offload APIs?

- Capture loop bodies in a (variadic) macro
  - Here variadic = "comma safe"

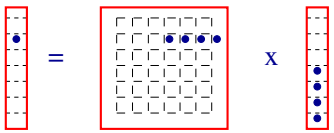
  https://github.com/paboyle/Grid/blob/develop/Grid/threads/Accelerator.h
- Grid uses the *accelerator_for* and *thread_for* macros to
- Think twice before using: *most* code should be using data parallel expression templates
- Hadrons contains (almost?) *no* cases of thread/accelerator for loops
- Interface is subject to change in future, so if you need it submit a function to Grid

How do you ensure contiguous accesses?

Unifying CPU and GPU programming models

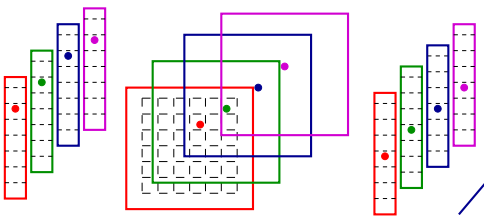
- Memory layout: both SIMD and SIMT need a vector direction
- Optimised stencil code will see lower-level interfaces
- SIMD and SIMT becomes exposed

Vector = Matrix x Vector



Reduction of vector sum
is bottleneck for small N

Many vectors = many matrices x many vectors

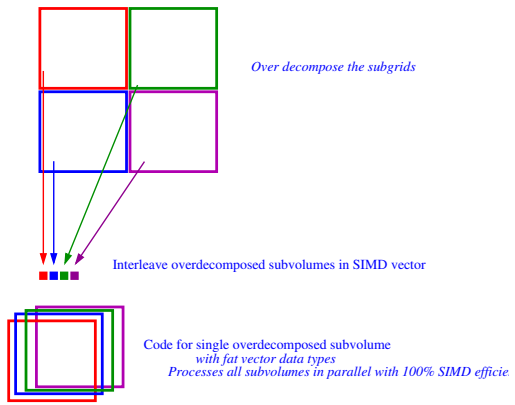


No reduction or SIMD lane
crossing operations.

SIMD interleave

How do you ensure contiguous accesses?

- Geometrically decompose cartesian arrays across nodes (MPI)
- Subdivide node volume into smaller *virtual nodes*
- Spread virtual nodes across SIMD lanes
- Use OpenMP+MPI+SIMD to process conformable array operations
- Same instructions executed on many nodes, each node operates on four virtual nodes



- Conclusion: Modify data layout to align data parallel operations to SIMD hardware
- Conformable array operations simple and vectorise perfectly

Must permute red/green/blue/purple when you access off the edge of sub-cell

How can you manage memory?

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**

- Software convention:
  - must obtain a *view* of an object for host/acccelerator from software cache manager - returns pointer to use
  - triggers data motion as required

Stencil implementation of free Laplacian

```
template<class Field> class FreeLaplacianStencil : public SparseMatrixBase<Field>
{
  typedef typename Field::vector_object siteObject;
  typedef CartesianStencil<siteObject, siteObject, int> StencilImpl;

  StencilImpl Stencil;
  SimpleCompressor<siteObject> Compressor;

  FreeLaplacianStencil(GridBase *_grid) : Stencil    (_grid,6,Even,directions,displacements,0), grid(_grid) {};

  virtual void  M    (const Field &_in, Field &_out)
  {
    Stencil.HaloExchange(_in, Compressor);     // Halo exchange for this geometry of stencil

    auto buf = st.CommBuf();
    auto st = Stencil.View(AcceleratorRead);       // Views; device friendly/accessible pointers
    autoView( in     , _in    , AcceleratorRead);
    autoView( out    , _out   , AcceleratorWrite);

    typedef typename Field::vector_object     vobj;
    typedef decltype(coalescedRead(in[0])) calcObj;
    const int      Nsimd = vobj::Nsimd();
    const uint64_t NN = grid->oSites();

    accelerator_for( ss, NN, Nsimd, {

        StencilEntry *SE;
        const int lane=acceleratorSIMTlane(Nsimd);
        calcObj chi, res;

#define LEG_LOAD(Dir)                                           \
  SE = st.GetEntry(ptype, Dir, ss);                             \
  if (SE->_is_local ) {                                         \
    int perm= SE->_permute;                                     \
    chi = coalescedReadPermute(in[SE->_offset],ptype,perm,lane); \
  } else {                                                      \
    chi = coalescedRead(buf[SE->_offset],lane);                 \
  }                                                             \
  acceleratorSynchronise();

        res                 = coalescedRead(in[ss])*(-6.0);
        LEG_LOAD(0); res = res + chi;
        LEG_LOAD(1); res = res + chi;
        LEG_LOAD(2); res = res + chi;
        LEG_LOAD(3); res = res + chi;
        LEG_LOAD(4); res = res + chi;
        LEG_LOAD(5); res = res + chi;

        coalescedWrite(out[ss], res,lane);
    });
  };
};
```

Stencil implementation of covariant Laplacian

```
template<class Gimpl,class Field> class CovariantLaplacianStencil : public SparseMatrixBase<Field>
{
  StencilImpl Stencil;
  SimpleCompressor<siteObject> Compressor;
  DoubledGaugeField Uds;

  CovariantLaplacianStencil(GaugeField &Umu) : grid(Umu.Grid()), Stencil(grid,6,Even,directions,displacements,0), Uds(grid)
  {
    for (int mu = 0; mu < Nd; mu++) {
      auto U = PeekIndex<LorentzIndex>(Umu, mu);
      PokeIndex<LorentzIndex>(Uds, U, mu );
      U = adj(Cshift(U, mu, -1));   // Double store gauge field (8 links per site)
      PokeIndex<LorentzIndex>(Uds, U, mu + 4);
    }
  };

  virtual void  M    (const Field &_in, Field &_out)
  {
    Stencil.HaloExchange(_in, Compressor);// Halo exchange for this geometry of stencil

    // Arithmetic expressions
    auto st = Stencil.View(AcceleratorRead);
    auto buf = st.CommBuf();
    StencilEntry *SE;

    autoView( in     , _in    , AcceleratorRead);
    autoView( out    , _out   , AcceleratorWrite);
    autoView( U      , Uds    , AcceleratorRead);

    typedef typename Field::vector_object        vobj;
    typedef decltype(coalescedRead(in[0]))    calcObj;
    typedef decltype(coalescedRead(U[0](0))) calcLink;

    const int      Nsimd = vobj::Nsimd();
    const uint64_t NN = grid->oSites();
    accelerator_for( ss, NN, Nsimd, {

        calcObj chi, res;
        calcLink UU;

#define LEG_LOAD_MULT(leg,polarisation) \
        UU = coalescedRead(U[ss](polarisation)); \
        LEG_LOAD(leg); \
        res = res + UU*chi();

        res                 = coalescedRead(in[ss])*(-6.0);
        LEG_LOAD_MULT(0,Xp);
        LEG_LOAD_MULT(1,Yp);
        LEG_LOAD_MULT(2,Zp);
        LEG_LOAD_MULT(3,Xm);
        LEG_LOAD_MULT(4,Ym);
        LEG_LOAD_MULT(5,Zm);

        coalescedWrite(out[ss], res,lane);
    });
  };
};
```

Exercise

- Implement a stencil based adjoint representation Laplacian for SU(3).
  - Check against the slow Cshift implementation:
    https://github.com/paboyle/Grid/blob/develop/Grid/qcd/utils/CovariantLaplacian.h

Scalar propagator

- $S = \phi^*(\Box + m^2)\phi$

$$M(x, x') = \delta_{x,x'} 2(N_d + m^2) - \sum_{\mu} \delta_{x+\mu,x'} + \delta_{x-\mu,x'}$$

- **Free case:** M is hermitian diagonalised by a unitary transformation

$$M = V^\dagger D V$$

- Call it the "discrete fourier transform" in the free case, and the eigenvalues are

$$D(p) = (2 \sin p/2)^2 + m^2$$

  - Propagator is the inverse of this

- **Interacting case:** M is hermitian diagonalised by a unitary transformation

$$M = V^\dagger D V$$

- covariant derivative couples to gauge fields, numerical solution of propagator
- Still diagonalisable, eigenvectors no longer plane waves etc...

$$M^{-1} = V \mathrm{Diag}\{\frac{1}{\lambda_i}\} V^\dagger \simeq V \mathrm{Diag}\{P(\lambda_i)\} V^\dagger$$

- If P is polynomial approximating $\frac{1}{x}$ over the whole spectral range $\Rightarrow$ Krylov solvers

Covariant smearing

- Base on free field Gaussian function:

$$S(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-x^2/2\sigma^2} \Leftrightarrow \tilde{S}(p) = \frac{1}{\sqrt{2\pi}} e^{-p^2\sigma^2/2}$$

- Approximate with fixed a polynomial:
  - $e^{-x} \sim (1 - \frac{x}{N})^N = \sum_n (N|n)(\frac{x^n}{N^n}) = \sum_n \frac{N!}{N^n(N-n)!} \frac{1}{n!} x^n \to \sum_n \frac{x_n}{n!}$
- Where the limit is true term by term in the series; convergence will occur for fixed x with large N
- Wuppertal smearing: apply this polynomial of covariant Laplacian

$$\mathbb{P}(L)(x, y) \sim e^{|x-y|^2/2\sigma^2}$$

  - It will convolve an input vector $b(y)$ with the Gaussian in the free field case
  - Interacting case is the gauge covariant generalisation
- Uses:
  - $b(y) = \delta_{y,y_0}$: Gaussian source at y_0
  - $b(y) = \delta_{y_t,t_0} \mathbb{Z}_2(y)$: (stochastic) Wall of Gaussian sources
  - Sink smearing of a propagator
  - Sequential sources for 3point functions, semileptonic form factors

Covariant smearing

Example implementation:
https://github.com/paboyle/Grid/blob/develop/Grid/qcd/utils/CovariantSmearing.h

```
// Free field iterates
//   chi = (1 - w^2/4N p^2)^N chi
Real coeff = (width*width) / Real(4*Iterations);

for(int n = 0; n < Iterations; ++n) {
  Laplacian.M(chi,psi);
  chi = chi + coeff*psi;
}
```

Chebyshev polynomials & approximation

https://github.com/paboyle/Grid/blob/develop/Grid/algorithms/approx/Chebyshev.h

- Chebyshev polynomials provide a general scheme for polynomial approximation over a given range
- Exponentially accurate in polynomial order
- Numerical Recipes in "C", chapter 5-8.
- Rational approximation chapter 5-13

http://numerical.recipes/book/book.html

- Can apply to any Hermitian matrix (not just function of one variable!)
- Disadvantage: must know upper and lower spectral bounds - not a *black box method*

```
void operator() (LinearOperatorBase<Field> &Linop, const Field &in, Field &out)
```

Chebyshev polynomials & approximation

Orthogonal univariate polynomials on $[-1, 1]$

$$T_n(x) = \cos(n \arccos x)$$

Recursion relation:

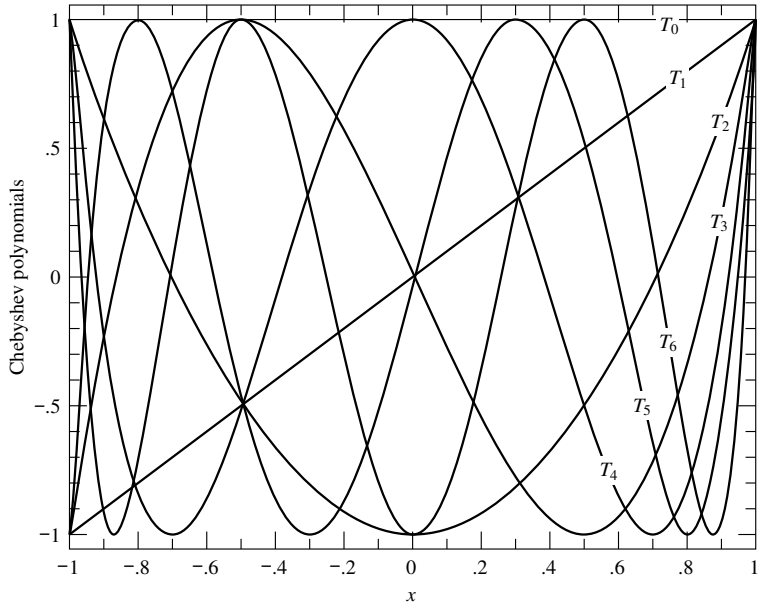
$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ &\cdots \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \quad n \geq 1. \end{aligned}$$

Chebyshev expansion coefficients:

$$\begin{aligned} c_j &= \frac{2}{N} \sum_{k=0}^{N-1} f(x_k) T_j(x_k) \\ &= \frac{2}{N} \sum_{k=0}^{N-1} f\left[\cos\left(\frac{\pi(k+\frac{1}{2})}{N}\right)\right] \cos\left(\frac{\pi j(k+\frac{1}{2})}{N}\right) \end{aligned}$$

Function approximation is exponentially accurate in order:

$$f(x) \approx \left[\sum_{k=0}^{N-1} c_k T_k(x)\right] - \frac{1}{2}c_0$$



- Special case of $f(x) = \frac{1}{x}$ corresponds to sparse matrix inversion
- Could used to solve propagators in LQCD (instead use Krylov solvers)
- Apply polynomial of matrix to a *source*
- Obtains one row of the inverse matrix
- Solving O(volume) times is required for complete inverse - impractical
- Never solve linear equations exactly; 10^{11} degrees of freedom
- 10^{-10} is typically sufficient