WP3 : Languages and Programming Models

WP3 working group:
Academic: Peter Boyle, Francesco di Renzo, Denis Barthou, and John Ashley
Codeplay: Paul Keir, Andrew Richards
Nvidia: John Ashley

- WP3 discussion report
- Data parallel programming
- Relation to work packages and communities

Machine landscape

- Understanding future progamming models and languages requires understanding future hardware
- Next generation US machines will mix GPU/simd and CPU/simd architectures
  e.g. OpenPower, ManyCore/Cray likely set the directions for large installations

	Chip	Insns	Width	single FP/clock/core
Xeon	Westmere	SSE2	128b	8
	Ivybridge	AVX	256b	16
	Haswell	AVX2	256b	32
	Broadwell 2015	AVX2	256b	32
	Skylake 2016	AVX3.2	512b	64
Many core	Knights Corner	AVX512	512b	32
	Knights Landing	AVX3.2	512b	64
GPU	Many SIMD cores	Warp scheduling		

- SIMD issues same intruction to multiple operands in a register file
- SIMD loads accelerated by accessing consecutive memory addresses
- Underlying gain from spatial locality of *memory reference* and *operations*
- CPU SIMD vectorises "data" but NOT "thread state"
  Single address and wide data load (spatial locality, all data on bus used)
  Lower performance gather operations where reference locality is absent
- GPU SIMD vectorises "data" AND "thread state"
  Many addresses generated
  Default is gather-like operation, spatial locality detected and *read coalescence* gives speed up
- *of course* thread and task parallelism must also be addressed.

Spatial locality

- Generating memory reference and operation spatial locality underlies optimisation for *all* next generation machines.
- Placing data upon which the same operations are performed in adjacent memory locations is often only possible to generate with data layout transformations.
- Current languages make guarantees about the data layout of arrays which cannot be changed
- We must create layout opaque containers for arrays so that these contracts may be broken
- This was done implicitly in classic SIMD languages such CMfortran, APE/Tao (three cheers for INFN!) etc..
- Much discussion in WP3 has centred on automating layout transformation, and providing platform neutral interfaces to SIMD.

Assumptions

Two Skype telecon meetings discussing framework.
We have achieved some consensus on certain assumptions

1. We believe the future of HPC as a whole will be heterogeneous in terms of architectures and performance of system components, although individual installations may be homogenous or heterogenous.
2. We believe that parallelism at all levels of code and system will become more and more critical for performance. This includes levels including machines, cores, and vector units.
3. We believe that given the scale of the problems the Center of Excellence is interested in, codes must be able to be adapted to take maximum advantage of a particular installations mix of hardware, storage, and networking capacity and capabilities.
4. We believe codes should be portable and adaptable to an installation with reasonable effort.
5. We believe that we cannot pick a winner nor should we seek to find a one size fits all answer in terms of programming technologies such as CUDA, OpenCL, SYCL, OpenMP, OpenACC, etc.

Recommended directions for programming models

1. The organization and management of data is fundamental to the ability to organize and parallelize programs. Data structures should be fundamentally data parallel. In particular, they should be fundamentally vectorization friendly.
2. Parallel data structures should be able to be customized (at least at compile time) to underlying hardware without impacting the overall structure of the code.
3. Of the current languages used for high performance and high productivity programming, we feel C++ has the best chance of providing scalable, cross platform, and high performance code in a mature and still growing platform.
4. Fundamental implementations of algorithmic parallelism should be adaptable at compile time to varying underlying hardware or software architectures of a given installation. Distribution and balancing of workload between system components may be either done at compile time or runtime.

Recommended areas for study

1. Evaluation of various data parallel structures such as Structure of Arrays, Array of Structures of Vectors, and potentially others. Develop examples of the use of each for common programming tasks and compare efficiency and ease of use.
2. Investigate use of ifdefs[1] , macros, templates, expression templates, and other language abstractions for abstracting algorithmic parallelism from written code while allowing reasonable compilation of performant code.
3. Development of a style guide for using techniques to enable varied physical implementations of abstracted parallelism allowing for the incorporation of hand coded routines using existing compilers. The overall research program should strive to include examples for OpenMP, OpenACC, CUDA, and OpenCL/SYCL at a minimum, but individual investigators may not cover all these areas.
4. In conjunction with other work packages, examine weaknesses and strengths of existing Extreme Scale codes and look for functions and design patterns that can be added to a standard cookbook and/or library. Evaluate the benefits of embedded Domain Specific Languages (DSLs) from a programmer productivity vs algorithmic flexibility vs executable performance perspective. As an example, we could study QDP++ for lessons learned and opportunities for expansion and performance improvement.

We do not have a consensus view on the inclusion of areas beyond lattice physics and perhaps cosmology although there would be support for including CFD in the Lattice Boltzman formulation as it is similar to LQCD. We recognize there may be some benefits to including SKA and LHC.

[1] D. Barthou does not endorse ifdef

C++ data parallel objects?

Recreate success of CMfortran, APE/Tao for cartesian array processing?

- Automating layout transformation is way forward
- Conformable array operations automatically map to independent threads and independent SIMD lanes.
- Proof of concept in C++ container library
- Object oriented SIMD interface – perhaps in spirit of APE/Tao?

Connection virtual nodes $\Rightarrow$ independent subvolumes in independent SIMD lanes

Automating layout transformations

Ordering	Layout	Vectorisation	Data Reuse
Microprocessor	Array-of-Structs (AoS)	Hard	Maximised
Vector	Struct-of-Array (SoA)	Easy	Minimised
Bagel	Array-of-structs-of-short-vectors (AoSoSV)	Easy	Maximised

- Developed general short vector classes, compile time determined width.
- Parameterise transformation, opaque containers hide layout from user
- Automatically transform layout of mathematical objects (scalar, vector, matrix, higher rank tensors).

```
template<class vtype> class iScalar
{
    vtype _internal;
};
template<class vtype,int N> class iVector
{
    vtype _internal[N];
};
template<class vtype,int N> class iMatrix
{
    vtype _internal[N][N];
};
```

Data parallel programming interface

- Define matrix, vector, scalar operations using usual C++ elegance
  - Can nest `iMatrix<iMatrix<vfcomplex> >` etc... to form any tensor structure
- Can *change* the width of the internal type to short vectors *or* scalars
  - `CartArrayMatrix` $\rightarrow$ `CartArray<iMatrix>` $\rightarrow$ `vector<Matrix<vfcomplex> >`
  - `Matrix` $\longrightarrow$ `iMatrix<complex<float> >`
- Pass grid information in constructur
  - `CartArrayColorMatrix MyArray(Grid)`
- Conformable operations are data parallel on the *same* Grid layout
  - `HisArray = MyArray * YourArray;`
  - ```
    DerivativeArray = 0.5*(Cshift(OtherArray,2,-1 ) - Cshift(OtherArray,2,+1));
    ```
- Bottom line: high-level data parallel code gets 65% of peak
- Single data parallelism model targets BOTH SIMD and threads efficiently.
- Plan: cover MPI + SIMD + Threads in single parallelism model for cartesian array processing
    ```

Implementation details

Define performant classes *vfloat*, *vdouble*, *vfcomplex*, *vzcomplex*.

```
#if defined (AVX1) || defined (AVX2)
    typedef __m256 dvec;
#endif
#if defined (SSE2)
    typedef __m128 dvec;
#endif
#if defined (AVX512)
    typedef __m512 dvec;
#endif
#if defined (QPX)
    typedef vector4double dvec;
#endif
    class vdouble  {
         dvec v;
      // Define arithmetic operators
        friend inline vdouble operator + (vdouble a, vdouble b);
        friend inline vdouble operator - (vdouble a, vdouble b);
        friend inline vdouble operator * (vdouble a, vdouble b);
        friend inline vdouble operator / (vdouble a, vdouble b);
        static int Nsimd(void) { return sizeof(dvec)/sizeof(double);}
    }
```

Implementation details

Define performant classes *vfloat*, *vdouble*, *vfcomplex*, *vzcomplex*.

```
   friend inline vdouble operator + (vdouble a, vdouble b)  {
     vdouble ret;
#if defined (AVX1)|| defined (AVX2)
          ret.v = _mm256_add_pd(a.v,b.v);
#endif
...
     return ret;
    };

    friend inline vdouble operator * (vdouble a, vdouble b) {
      vdouble ret;
#if defined (AVX1)|| defined (AVX2)
      ret.v = _mm256_mul_pd(a.v,b.v);
#endif
...
      return ret;
    };

   friend inline void fmac (vdouble &y,vdouble a, vdouble x){
#if defined (AVX1) || defined (SSE2)
     y = a*x+y;
#endif
#ifdef AVX2     // AVX 2 introduced FMA support. FMA4 eliminates a copy, but AVX only has FMA3
     // accelerates multiply accumulate, but not general multiply add
     y.v = _mm256_fmadd_pd(a.v,x.v,y.v);
#endif
   }
```

Code examples & performance analysis

Matrix multiply is simply coded. Syntactic sugar connects to arithmetic operators.

```
template<class rrtype,class ltype,class rtype,int N>
inline void mult(iMatrix<rrtype,N> * __restrict__ ret,iMatrix<ltype,N> * __restrict__ lhs,iMatrix<rtype,N> * __restrict__ rhs){
    for(int c2=0;c2<N;c2++){
    for(int c1=0;c1<N;c1++){
        mult(&ret->_internal[c1][c2],&lhs->_internal[c1][0],&rhs->_internal[0][c2]);
        for(int c3=1;c3<N;c3++){
            mac(&ret->_internal[c1][c2],&lhs->_internal[c1][c3],&rhs->_internal[c3][c2]);
        }
    }}
    return;
}
```

- Template parameter matrix size; known at compile time
- Generates *very* efficient AVX/AVX2 code with clang

Clang++ Assembly output

```
Ltmp4:
        .cfi_def_cfa_register %rbp
        vmovaps (%rdx), %ymm0
        vmovaps 32(%rdx), %ymm1
        vmovaps 64(%rdx), %ymm2
        vmovaps 96(%rdx), %ymm3
        vmovaps 128(%rdx), %ymm4
        vmovaps 160(%rdx), %ymm5
        vmovaps 192(%rdx), %ymm6
        vmovaps 224(%rdx), %ymm7
        xorl    %eax, %eax
        .align  4, 0x90
LBB0_1:                                 ## %.preheader
                                        ## =>This Inner Loop Header: Depth=1
        vmulps  (%rsi,%rax,8), %ymm0, %ymm8
        vaddps  (%rdi,%rax), %ymm8, %ymm8
        vmulps  32(%rsi,%rax,8), %ymm1, %ymm9
        vaddps  %ymm9, %ymm8, %ymm8
        vmulps  64(%rsi,%rax,8), %ymm2, %ymm9
        vaddps  %ymm9, %ymm8, %ymm8
        vmulps  96(%rsi,%rax,8), %ymm3, %ymm9
        vaddps  %ymm9, %ymm8, %ymm8
        vmulps  128(%rsi,%rax,8), %ymm4, %ymm9
        vaddps  %ymm9, %ymm8, %ymm8
        vmulps  160(%rsi,%rax,8), %ymm5, %ymm9
        vaddps  %ymm9, %ymm8, %ymm8
        vmulps  192(%rsi,%rax,8), %ymm6, %ymm9
        vaddps  %ymm9, %ymm8, %ymm8
        vmulps  224(%rsi,%rax,8), %ymm7, %ymm9
        vaddps  %ymm9, %ymm8, %ymm8
        vmovaps %ymm8, (%rdi,%rax)
        addq    $32, %rax
        cmpq    $256, %rax              ## imm = 0x100
        jne     LBB0_1
```

- Template parameter matrix size (8) ; known at compile time
- Generates *very* efficient AVX/AVX2 code with clang

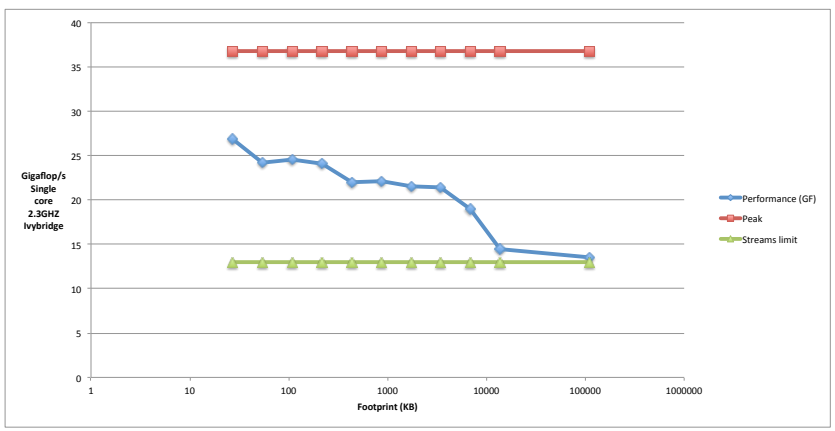
Performance analysis on Ivybridge

- FP pipeline
  - dual issue 8 wide single precision 2.3GHz.
  - Peak 16x2.3 = 36.8 Gflop/s per core single
  - Peak 8x2.3 = 18.4 Gflop/s per core double
- Memory system
  - Streams bandwidth benchmark reports 13GB/s.
  - Peak memory bandwidth 25.6GB/s.
- L1 resident results (should saturate FP pipe)
  - matmul with N=12 : 32Gflop/s

```
std:vector<int> grid = { 8,8,8,8 };
std:vector<int> simd = { 1,1,2,2 };

CartesianGrid Grid(grid,simd);
CartArrayColorMatrix A(Grid);
CartArrayColorMatrix B(Grid);
CartArrayColorMatrix C(Grid);

A = B * C;
```



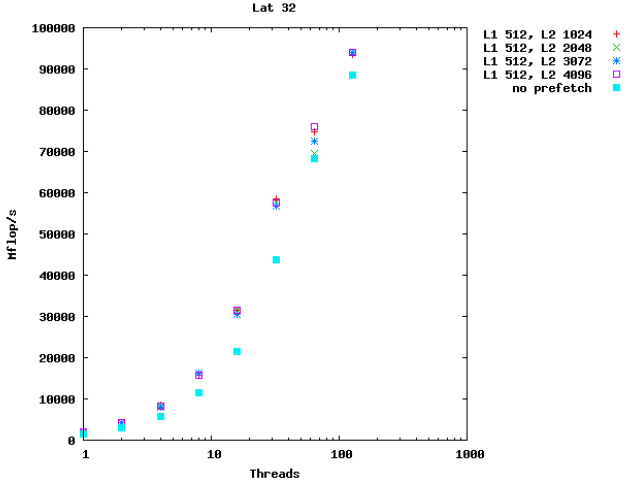
Performance on KNC

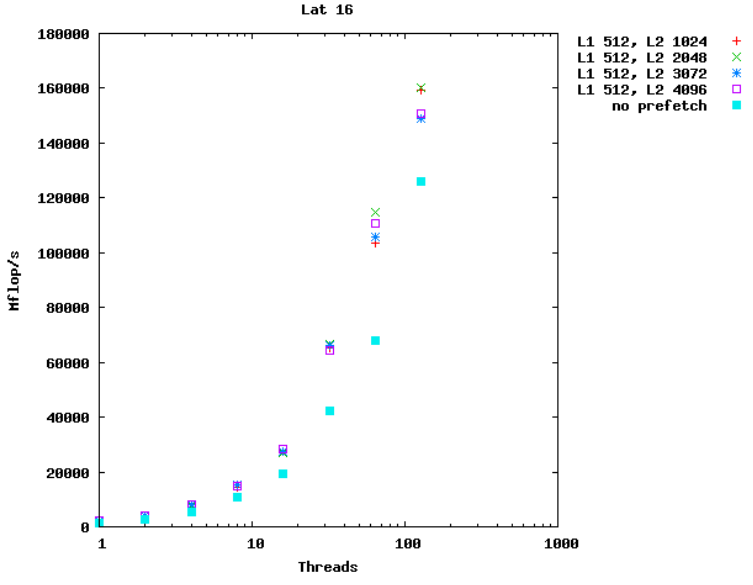
Added OpenMP threading to Lattice operations:

```
template<class left,class right>
inline auto operator - (Lattice<left> &lhs, right &rhs)
                -> Lattice<decltype(lhs._odata[0]*rhs)>
{
  Lattice<decltype(lhs._odata[0]*rhs)> ret(lhs._grid);
#pragma omp parallel for
  for(int ss=0;ss<rhs._grid->oSites(); ss++){
    ret._odata[ss]=lhs._odata[ss]-rhs;
  }
  return ret;
}
```

- Simple data parallel interface $\rightarrow$ SIMD $\otimes$ Threads $\otimes$ MPI
- Data parallel array operations and cshift
- Built on abstract vFloat, vComplex classes targetting SIMD intrinsics.
  SSE, SSE2, AVX, AVX2, AVX512, AVX3.2

Performance on KNC





OpenAcc thoughts

Probably quite easy with ifdefs.
Follow connection machine rule:

- scalar variables live on the host,
- vector variables live on the accelerator,
- use pcopyin to cache transparently.

John Ashley has verified that simple loop implementation of vFloat class does indeed vectorise on GPU through PGI compiler.
Alan Gray (EPCC) is playing a similar game with Lattice Boltzmann code

```
template<class left,class right>
inline auto operator - (Lattice<left> &lhs, right &rhs)
                -> Lattice<decltype(lhs._odata[0]*rhs)>
{
  Lattice<decltype(lhs._odata[0]*rhs)> ret(lhs._grid);
#ifdef GRID_OFFLOAD
#pragma acc pcopyin(lhs,rhs,ret)
#else
#pragma omp parallel for
#endif
  for(int ss=0;ss<rhs._grid->oSites(); ss++){
    ret._odata[ss]=lhs._odata[ss]-rhs;
  }
  return ret;
}
```

Performance analysis

Conclusion:

- Industry has a programming problem with proliferation of multiple forms of parallelism
  - SIMD $\otimes$ Threads $\otimes$ MPI
  - Current "solutions" such as co-arrays describe only a subset of these
- High performance data parallel interface is achievable through compiler
  - Accelerates regular cartesian array processing
- Simple data parallel interface $\rightarrow$ SIMD $\otimes$ Threads $\otimes$ MPI
- Obtain highest fractions of peak from high level code
  10x faster than QDP++ *and* 10x smaller than QDP++ *and* more general than QDP++ !

Suggestion how WP3 might be structured?

- Multi-layered interface to SIMD
  1. Portfolio of best practice examplars for a number of homogenous and heterogenous compilation targets OpenCL, SyCL, C++AMP, OpenAcc, OpenMP
     - Feeds into and informs application packages
  2. Cross-platform portable vector float/double/fcomplex/dcomplex type system
     - Scalar,SSE,SSE2,AVX,AVX2,AVX512,AVX3.2,QPX,GPU ready
  3. Generic performant mathematical types based on these: Scalar,Vector,Matrix; automating layout tranformation
     - Foundation for higher levels
     - Possibly useful to constructing some data parallel operations in codes that do not fit neatly into global regular structure (e.g. n-body simulation)
  4. Data-parallel Cartesian grid array processing library
     - Design target is QCD; possibly applicable to many COSMOS 3d grid algorithms such as WALLS, also UKMHD Lare3d code.

Relation between WP3 and other packages

- Initially PB supported a US SciDAC style model with complete code reengineering.
- Level of resource is insufficient to support reengineering of so many communities
  Evolving current packages leverages prior investments and WP3 should assist that

Serving the domains:

- WP3 appears presently most focussed on giving high performance to the easily parallelised QCD codes
- Degree to which it helps other communities is perhaps less clear

1. Provides examples of best practice programming techniques to the applications WP's
2. Document best practice use of SyCL, OpenACC, vector intrinsics
3. Layers may provide multiple points of entry usable code
   - PPE: Ensemble chisq min with vector types
   - Astro: Develop unstructured mesh interface using gather instructions?
   - SKA?
   - PPT: best practice performance programming examples,
     data parallel library may be used in packages
     PAB has ongoing "QPS" collaboration with BNL & Columbia;
     Successful application to NERSC/Cori early adoption programme (NESAP)

Discussion