Implementation of Scientific Computing Applications on the Cell Broadband Engine processor

Guochun Shi,

Volodymyr Kindratenko

National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign

Three scientific applications

- Nanoscale Molecule Dynamics (NAMD)
  - James Philips from Theoretical and Computation Biophysics group, UIUC
- MIMD Lattice Computation (MILC)
  - Steven Gottlieb from physics department, Indiana University
- Electron Repulsion Integral (ERI) in quantum chemistry
  - Ivan S. Ufimtsev, Todd J.Martinez, Chemistry department, UIUC

Presentation outline

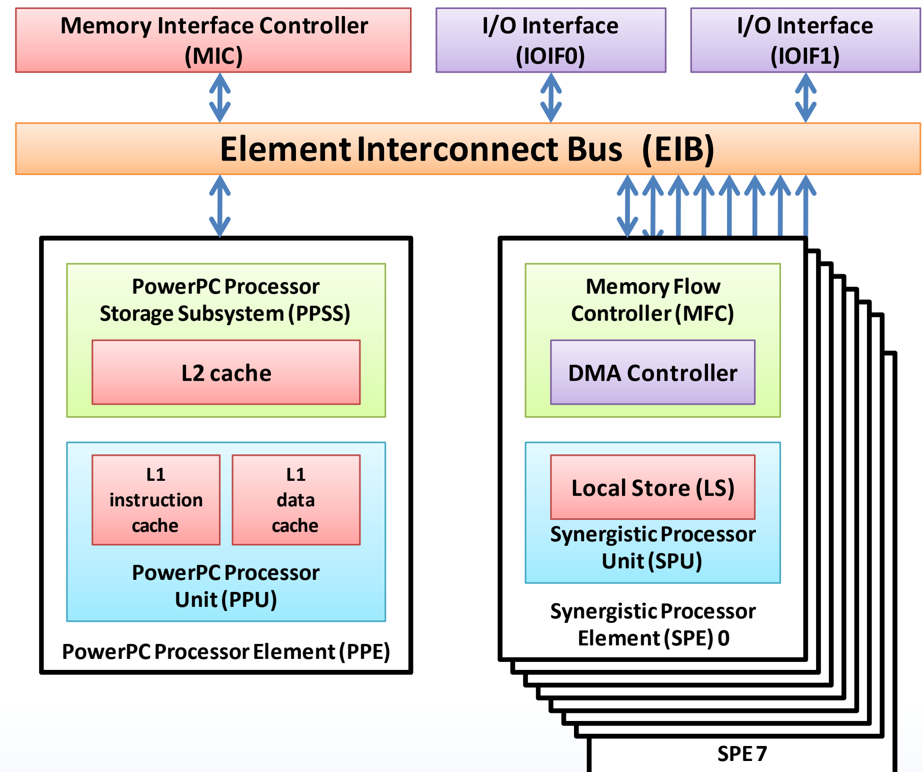
- Introduction
  - Cell Broadband Engine
  - Target Applications
    1. NAnoscale Molecule Dynamics (NAMD)
    2. MIMD Lattice Computation (MILC) collaboration
    3. Electron Repulsion Integral (ERI) in quantum chemistry
  - In-house task library and task dispatch system

- Implementation and performance
  - Application porting and optimization

- Summary
- Conclusions

Cell Broadband Engine

- One Power Processor Element (PPE) and eight Synergistic Processing Elements (SPE), each SPE has 256 KB local storage
- 3.2 GHz processor
- 25.6 GB/s processor-to-memory bandwidth
- > 200 GB/s EIB sustained aggregate bandwidth
- Theoretical peak performance of 204.8 GFLOPS (SP) and 14.63 GFLOPS (DP)



NCSA

In-house task library and dispatch system

Compute ta*sk* struct

```
typedef struct task_s {
      int cmd; // operand
      int size; // the size of task structure
} task_t;
```

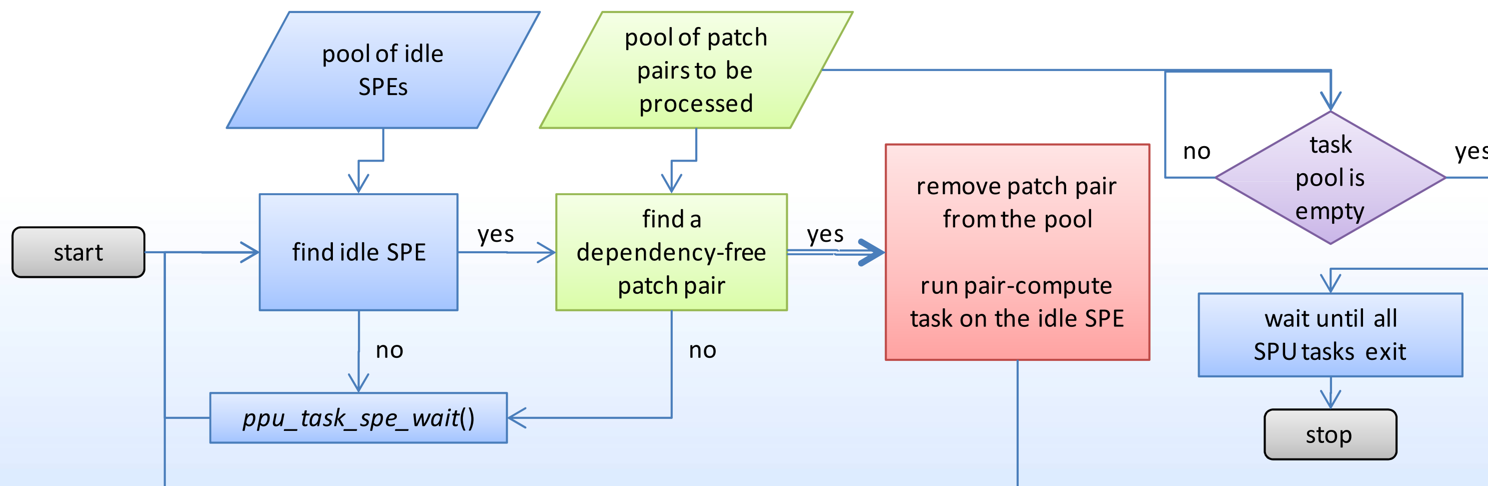
```
typedef struct compute_task_s {
      task_t common;
      <user_type1> <user_var_name1>
      <user_type2> <user_var_name2>
      …
} compute_task_t;
```

API for PPE and SPE

```
int ppu_task_init(int argc, char **argv,spe_program_handle_t );  // initialization
int ppu_task_run(volatile task_t * task);  // start a task in all SPEs
int ppu_task_spu_run(volatile task_t* task, int spe); // start a task in one SPE
int ppu_task_spu_wait(void); // wait for any SPE to finish, blocking call
void ppu_task_spu_waitall(void); // wait for all SPEs to finish, blocking all
```

```
int spu_task_init(unsigned long long);
int spu_task_register(dotask_t,int);  // register a task
int spu_task_run(void);  // start the infinite loop, wait for tasks
```

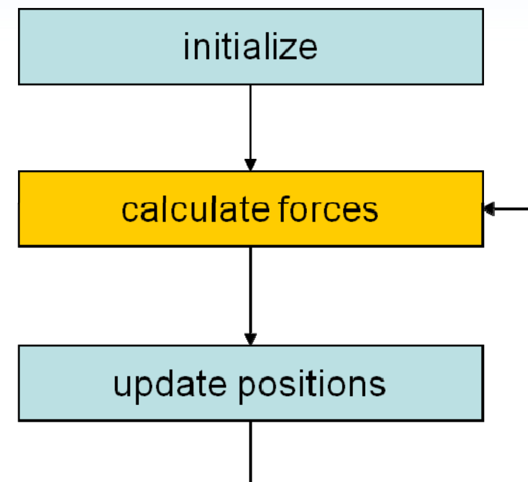
Task dispatch system in NAMD

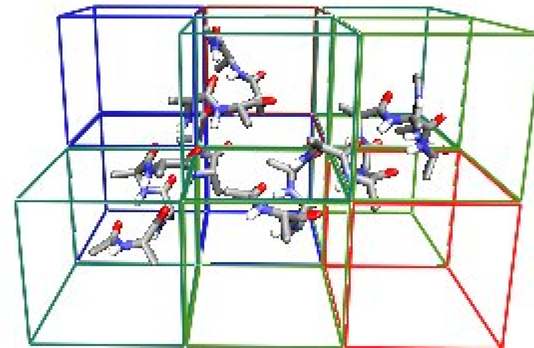


NCSA

NAMD

- Compute forces and update positions repeatedly
- The simulation space is divided into rectangular regions called patches
  - Patch dimensions > cutoff radius for non-bonded interaction
- Each patch only needs to be checked against nearby patches
  - Self-compute and pair-compute





NCSA

NAMD kernel

NAMD SPEC 2006 CPU benchmark kernel

- 1: for each atom *i* in patch p_k
- 2: for each atom *j* in patch p_l
- 3: if atoms *i* and *j* are bonded, compute bonded forces
- 4: otherwise, if atoms *i* and *j* are within the cutoff distance, add atom *j* to the *i*'s atom pair list
- 5: end
- 6: for each atom *k* in the *i*'s atom pair list
- 7: compute non-bonded forces (L-J potential and PME direct sum, both vial lookup tables)
- 8: end
- 9: end

We implemented a simplified version of the kernel that **excludes pairlists and bonded forces**

- 1: for each atom *i* in patch p_k
- 2: for each atom *j* in patch p_l
- 3: if atoms *i* and *j* are within the cutoff distance
- 4: compute non-bonded forces (L-J potential and PME direct sum, both via lookup tables)
- 5: end
- 6: end

NCSA

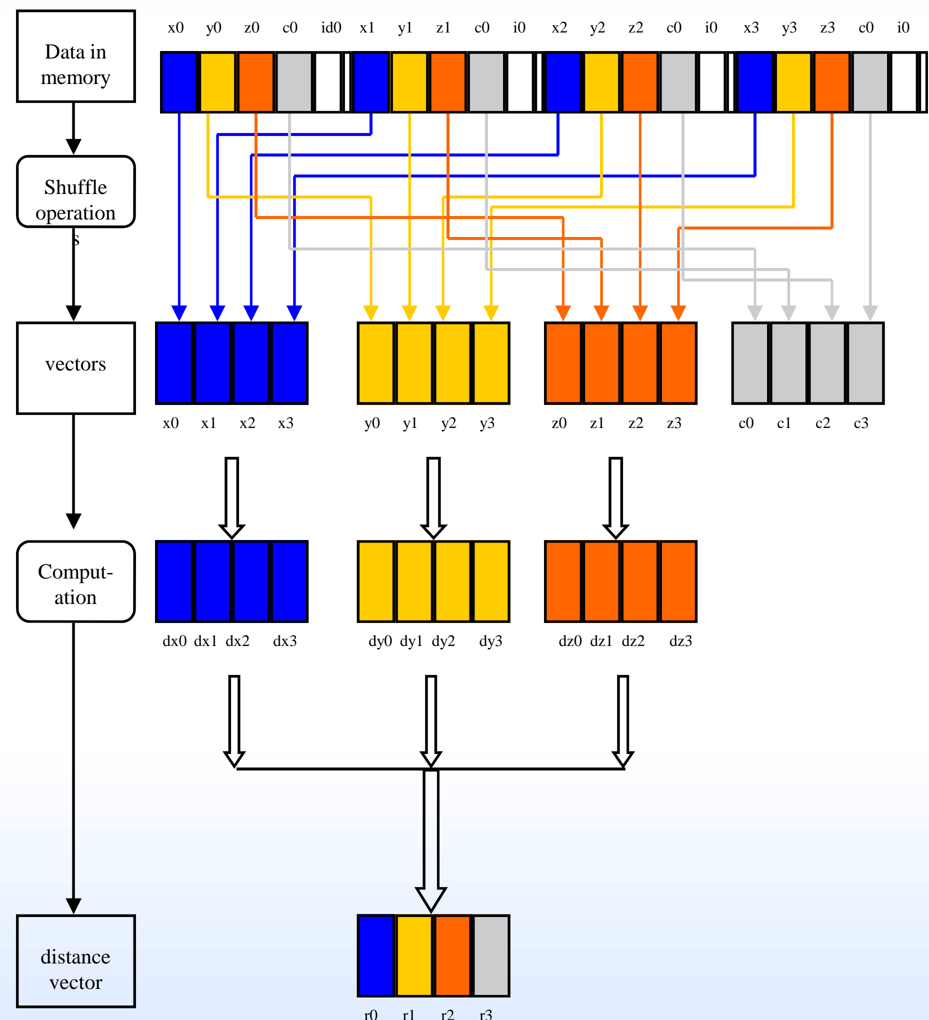
NAMD Implementation: SPU

- SIMD: each component is kept in a separate vector
- Data movement dominates the time
- Buffer size carefully chosen to fit into the local store

Local store usage

	Code (KB)	L-J table (KB)	Table _four (KB)	Atom_ buffer (KB)	Force buffer (KB)	Stack others
SP	25	55	45	30	18	83
DP	25	55	91	48	18	19

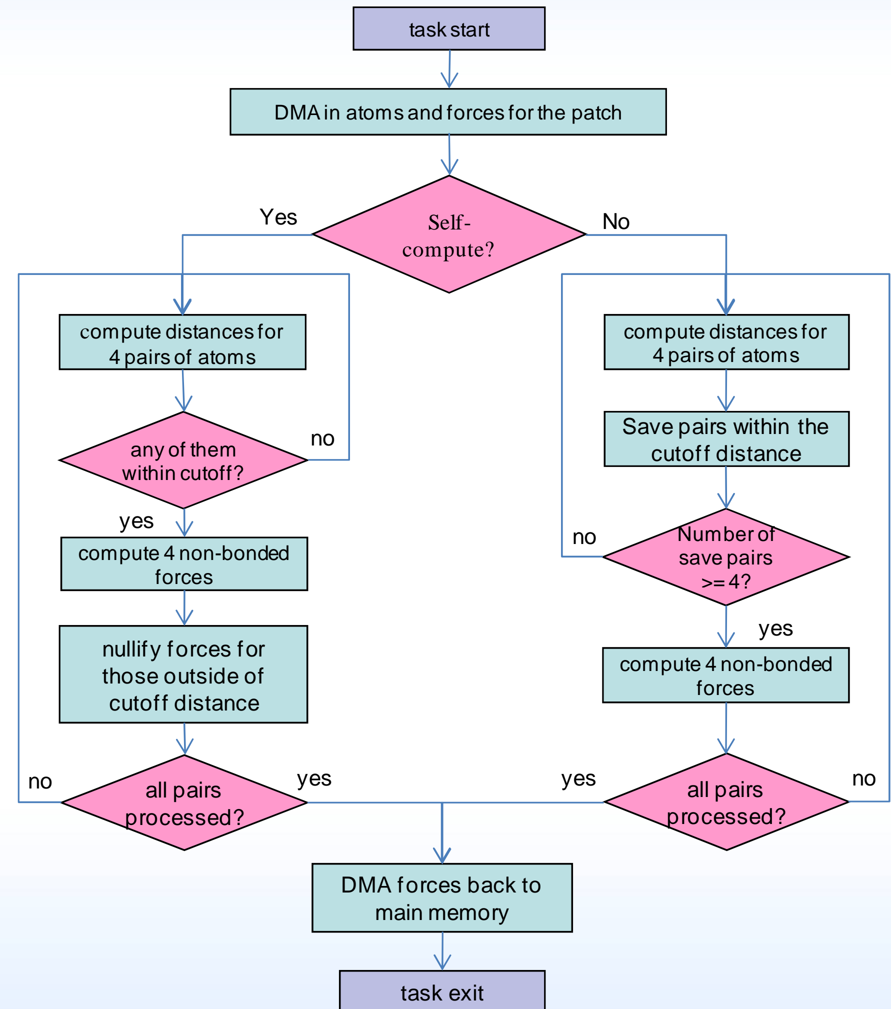
Data movement for distance computation (SP case)



NCSA

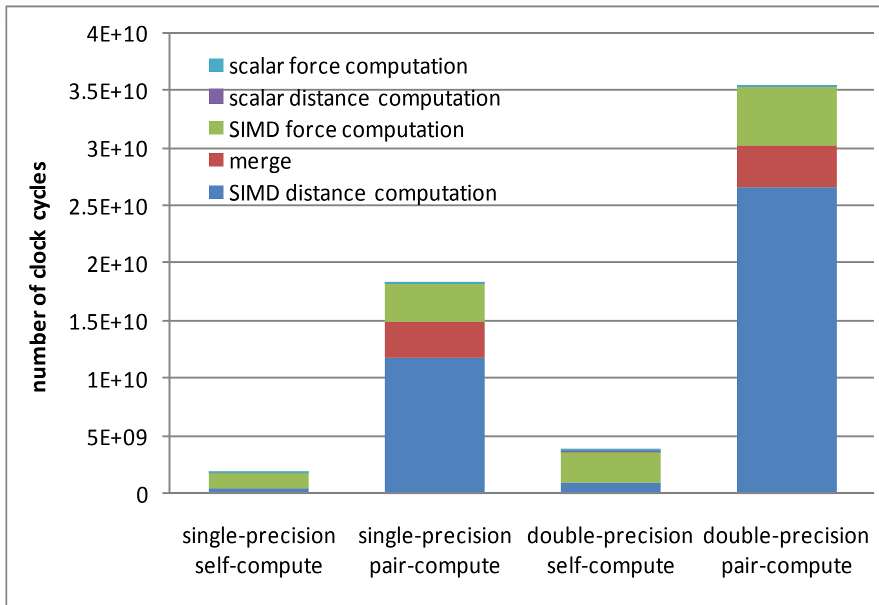
Implementation: optimizations

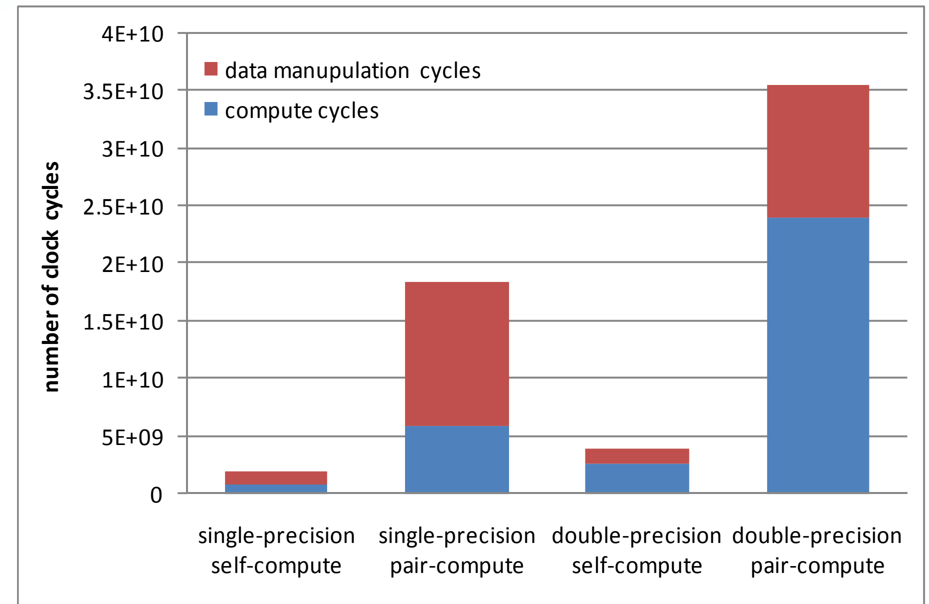
- Different vectorization schemes are applied in order to get best performance
  - Self-compute: do redundant computations and fill zeros to unused slots
  - Pair-compute: save enough pairs of atoms, then do calculations



NCSA

NAMD Performance: static analysis



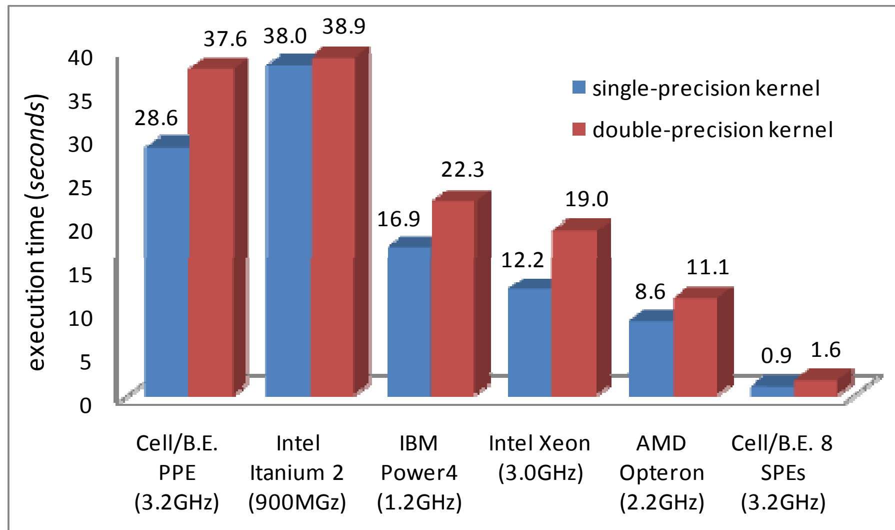


- Distance computation code takes most of the time
- Data manipulation time is significant

NCSA

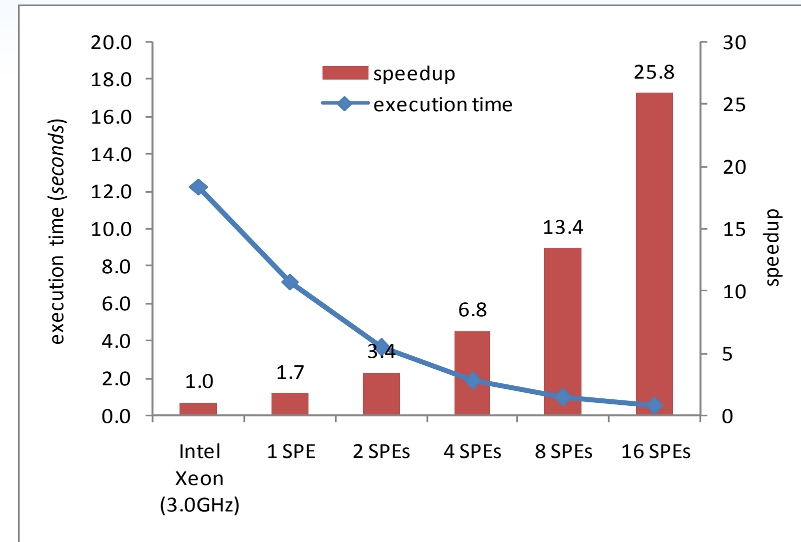
NAMD Performance

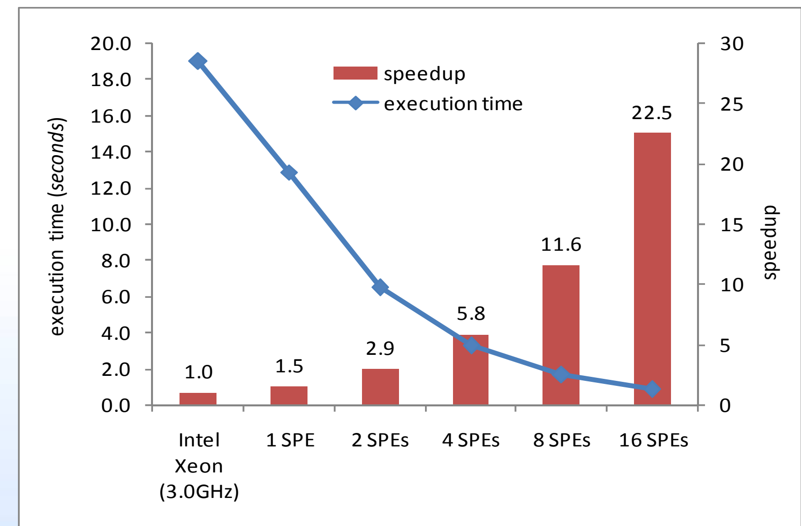
NAMD kernel performance on different architectures



- 13.4x speedup for SP and 11.6x speedup for DP compared to a 3.0 GHz Intel Xeon processor
- SP performance is < 2x better than DP

Scaling and speedup of the force-field kernel as compared to a 3.0 GHz Intel Xeon processor

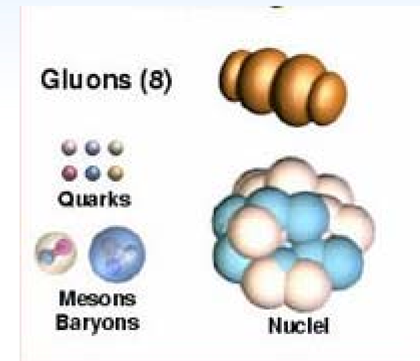


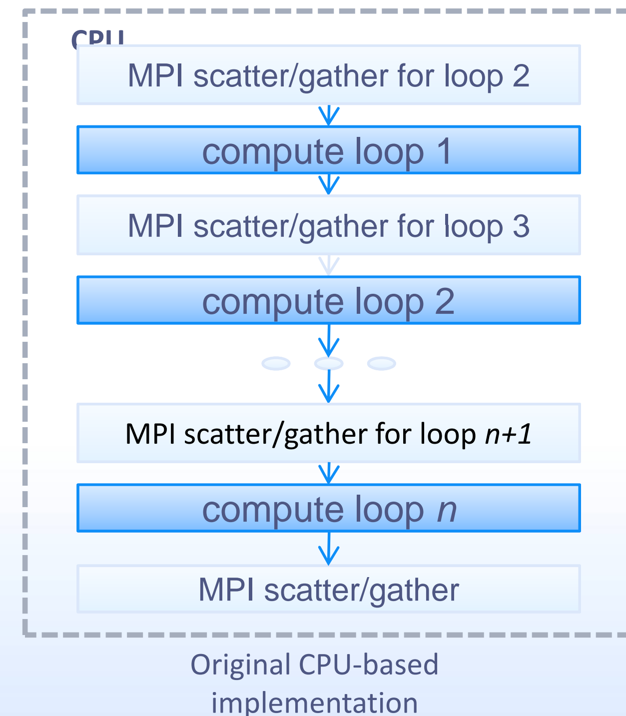


NCSA

MILC Introduction

- MILC studies large scale numerical simulations to study quantum chromodynamics (QCD), the theory of the strong interactions of subatomic physics.
- Large cycle consumer of supercomputers
- Our target application
  - dynamical clover fermions (clover_dynamical) using the hybrid-molecular dynamics *R* algorithm
- Our view of the MILC applications
  - A sequence of communication and computation blocks



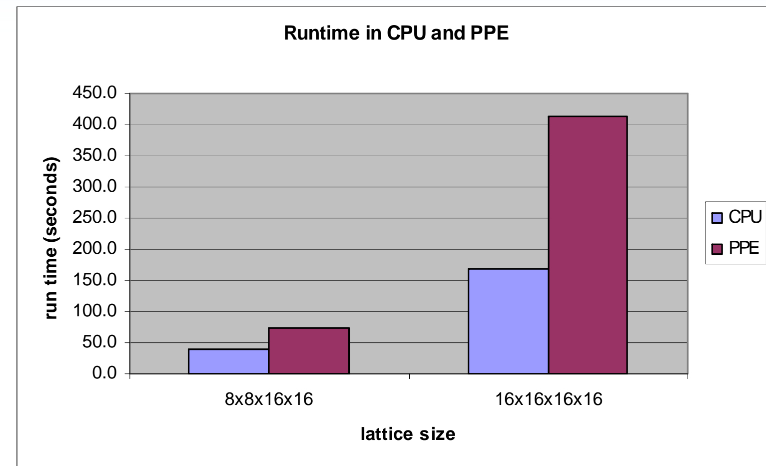


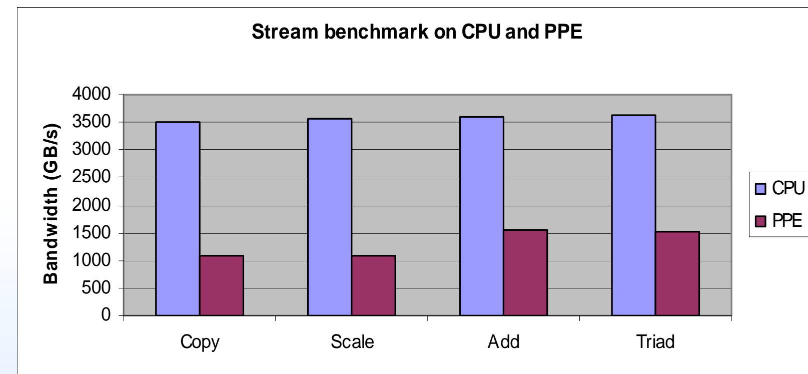
Original CPU-based implementation

NCSA

MILC Performance in PPE

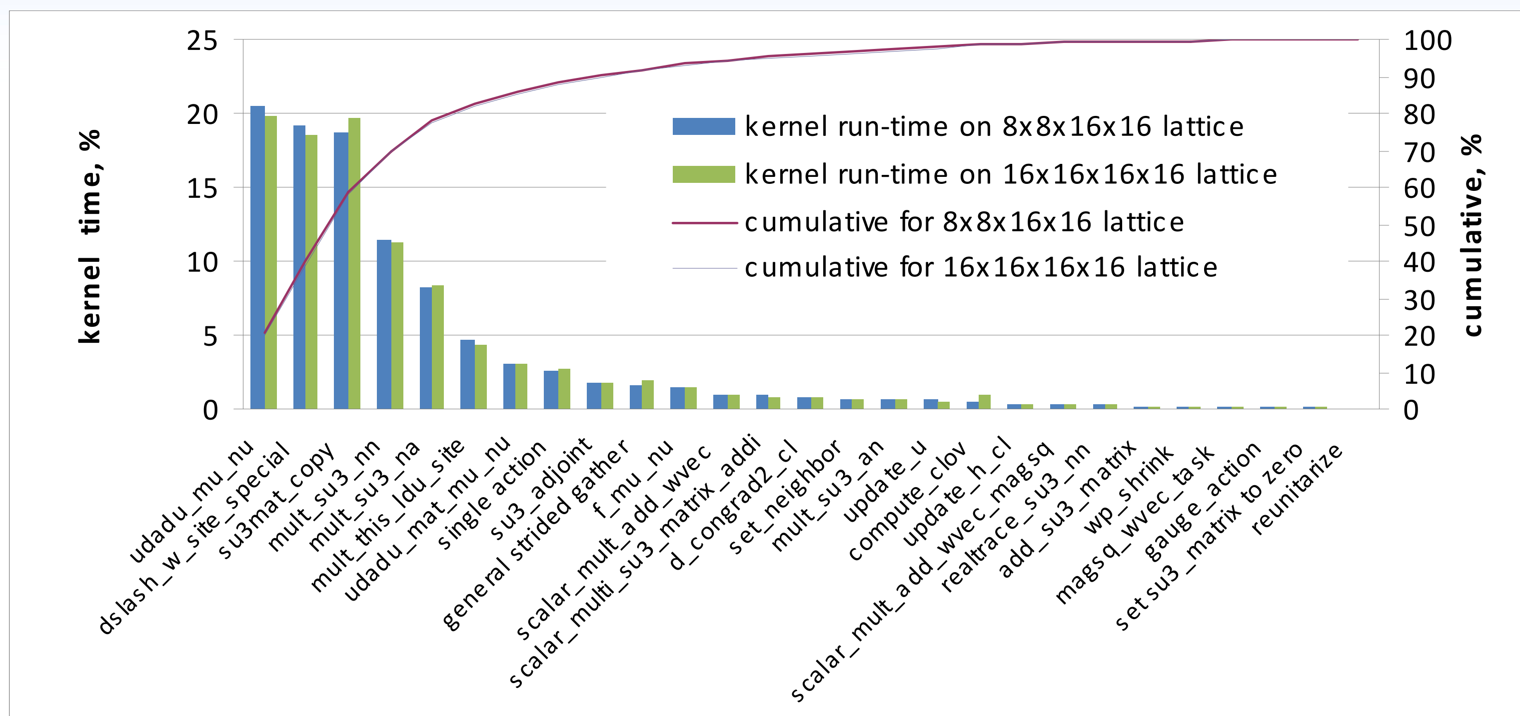
- Step 1: try to run it in PPE
- In PPE it runs approximately ~2-3x slower than modern CPU
- MILC is bandwidth-bound
- It agrees with what we see with stream benchmark





NCSA

MILC: Execution profile and kernels to be ported



- 10 of these subroutines are responsible for >90% of overall runtime
- su3mat_copy, i.e. memcpy responsible for nearly 20% of all runtime
- All kernels responsible for 98.8%

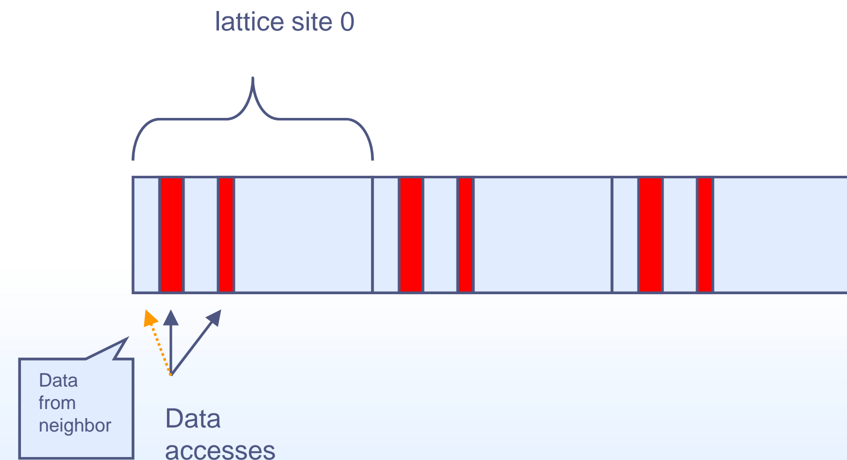
NCSA

MILC: Kernel memory access pattern

- Neighbor data access taken care of by MPI framework
- In each iteration, only small elements are accessed
  - Lattice size: 1832 bytes
  - su3_matrix: 72 bytes
  - wilson_vector: 96 bytes
- Challenge: how to get data into SPUs as fast as possible?
  - Data is nonaligned
  - Daa is not multiple of 128 bytes

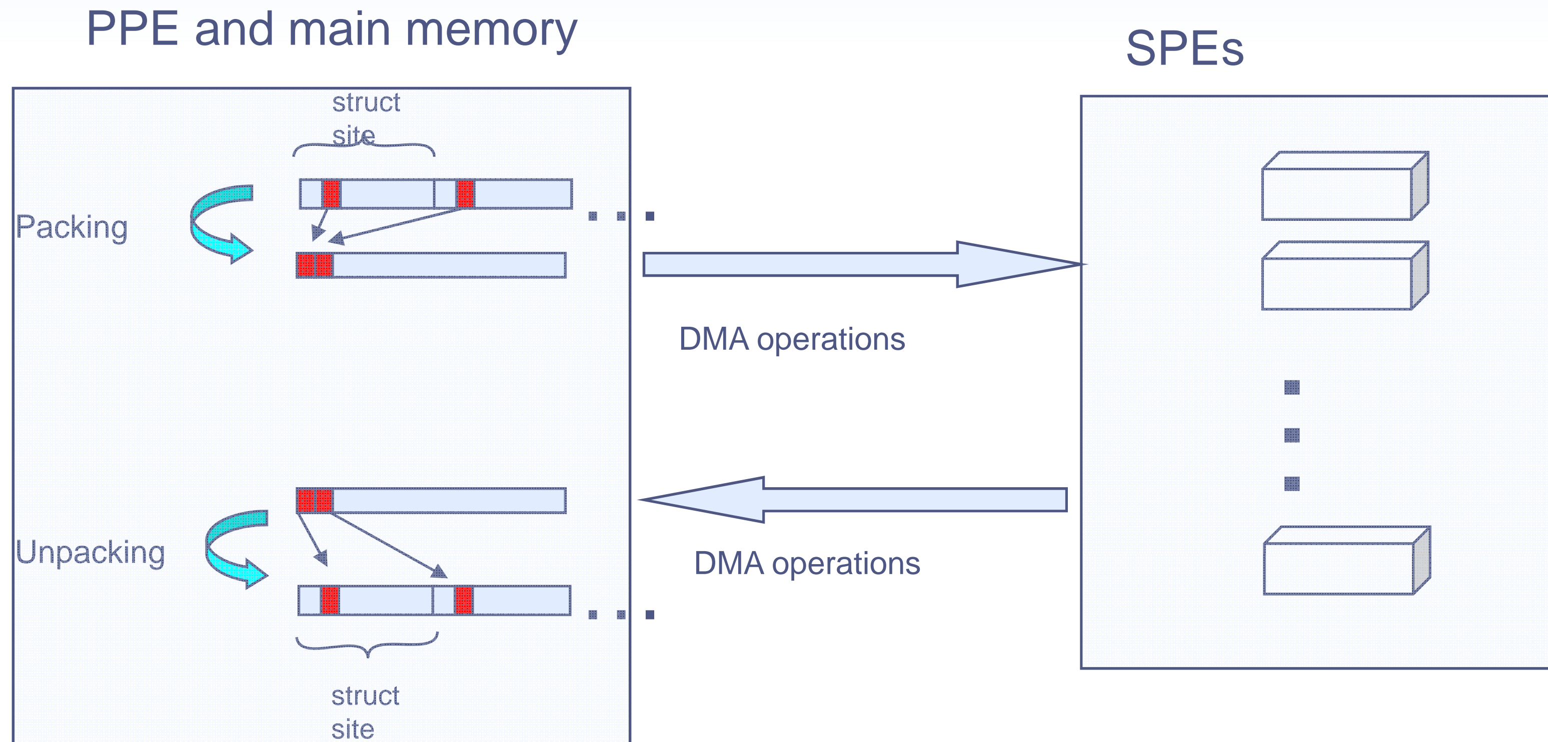
```
#define FORSOMEPARITY(i,s,choice) \
   for( i=((choice)==ODD ? even_sites_on_node : 0 ),  \
   s= &(lattice[i]); \
   i< ( (choice)==EVEN ? even_sites_on_node : sites_on_node); \
   i++,s++)

 FORSOMEPARITY(i,s,parity) {
      mult_adj_mat_wilson_vec( &(s->link[nu]), ((wilson_vector *)F_PT(s,rsrc)), &rtemp );
      mult_adj_mat_wilson_vec( (su3_matrix *)(gen_pt[1][i]), &rtemp, &(s->tmp) );
      mult_mat_wilson_vec( (su3_matrix *)(gen_pt[0][i]), &(s->tmp), &rtemp );
      mult_mat_wilson_vec( &(s->link[mu]), &rtemp, &(s->tmp) );
      mult_sigma_mu_nu( &(s->tmp), &rtemp, mu, nu );
      su3_projector_w( &rtemp, ((wilson_vector *)F_PT(s,lsrc)),
((su3_matrix*)F_PT(s,mat)) );
         }
```

One sample kernel from udadu_mu_nu() routine



NCSA

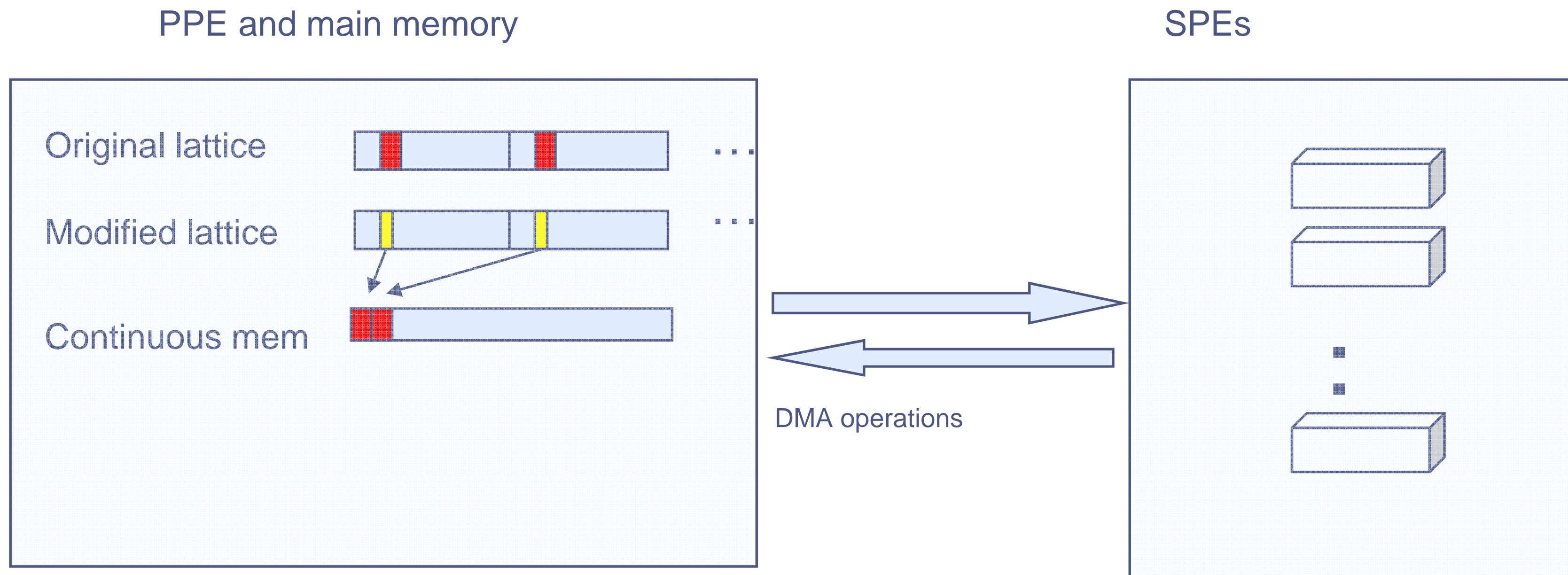
Approach I: packing and unpacking



- Good performance in DMA operations
- Packing and unpacking are expensive in PPE

NCSA

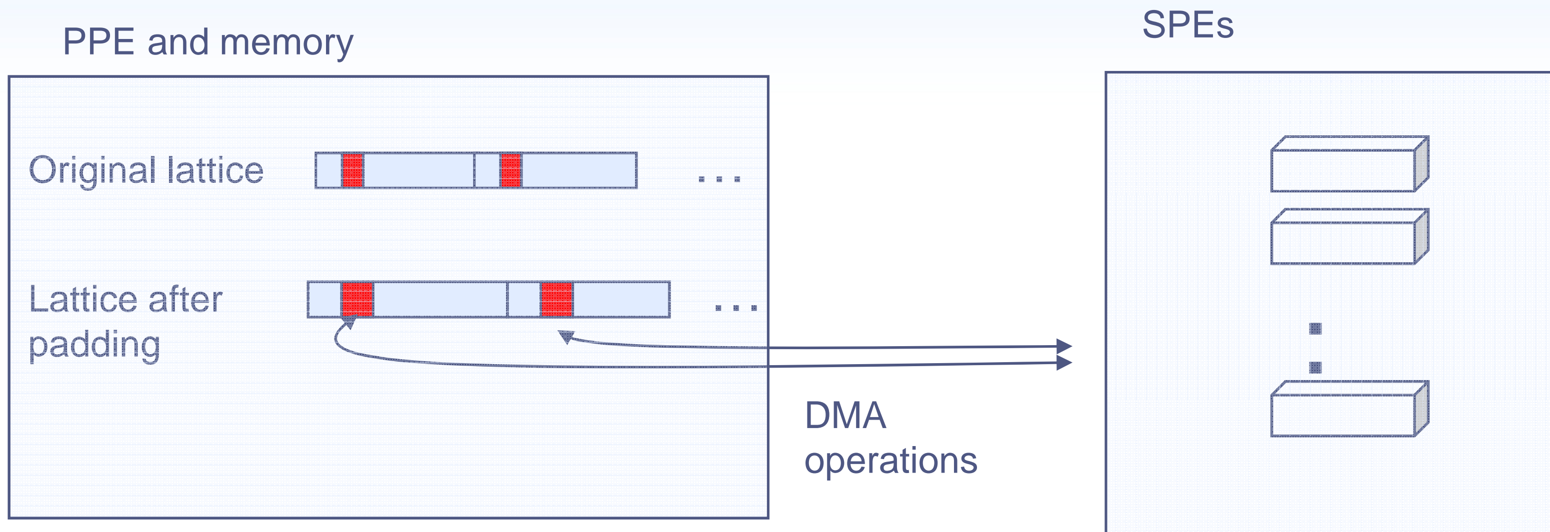
Approach II: Indirect memory access



- Replace elements in struct site with pointers
- Pointers point to continuous memory regions
- PPU overhead due to indirect memory access

NCSA

Approach III: Padding and small memory DMAs

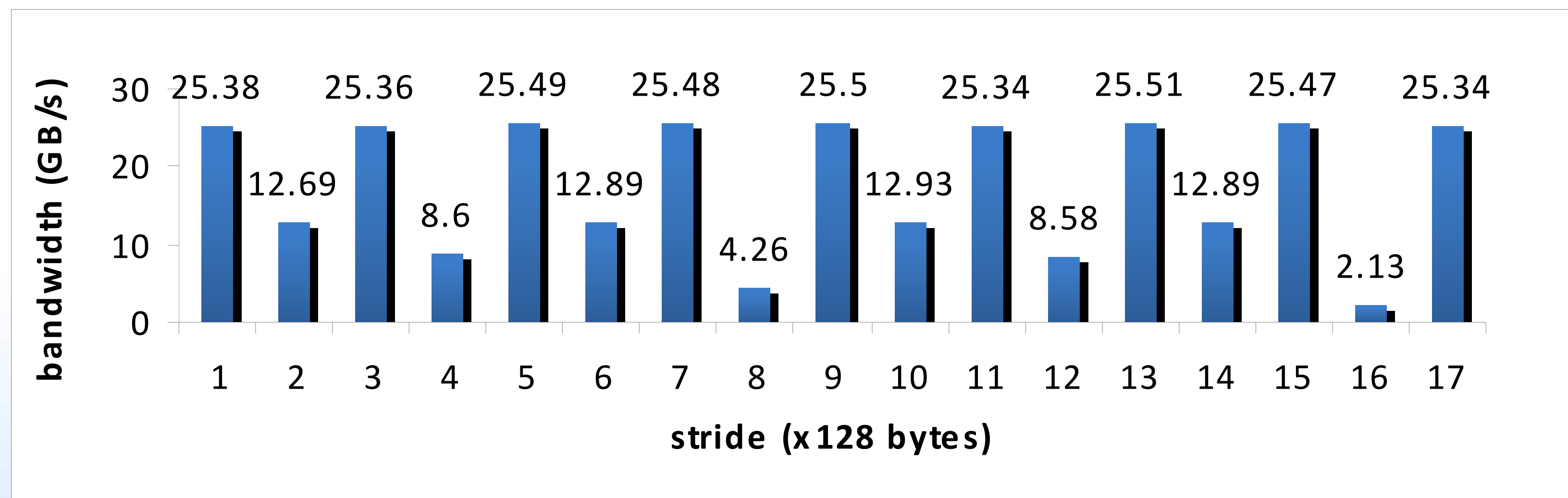


- Padding elements to appropriate size
- Padding struct site to appropriate size
- Gained good bandwidth performance with padding overhead
- Su3_matrix from 3x3 complex to 4x4 complex matrix
  - 72 bytes ➔ 128 bytes
  - Bandwidth efficiency lost: 44%
- Wilson_vector from 4x3 complex to 4x4 complex
  - 98 bytes ➔ 128 bytes
  - Bandwidth efficiency lost: 23%

NCSA

MILC struct site padding

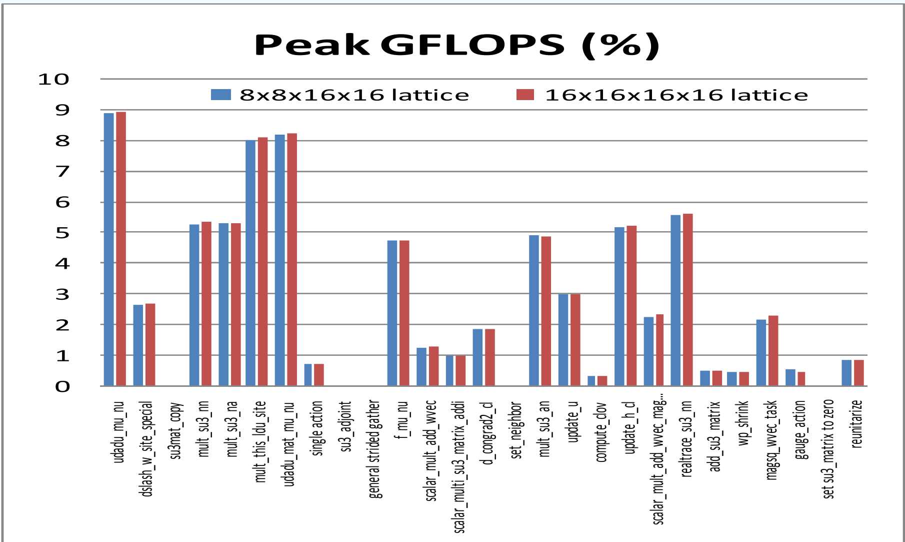
- 128 byte stride access has different performance
- This is due to 16 banks in main memory
- Odd numbers always reach peak
- We choose to pad the struct site to 2688 (21*128) bytes

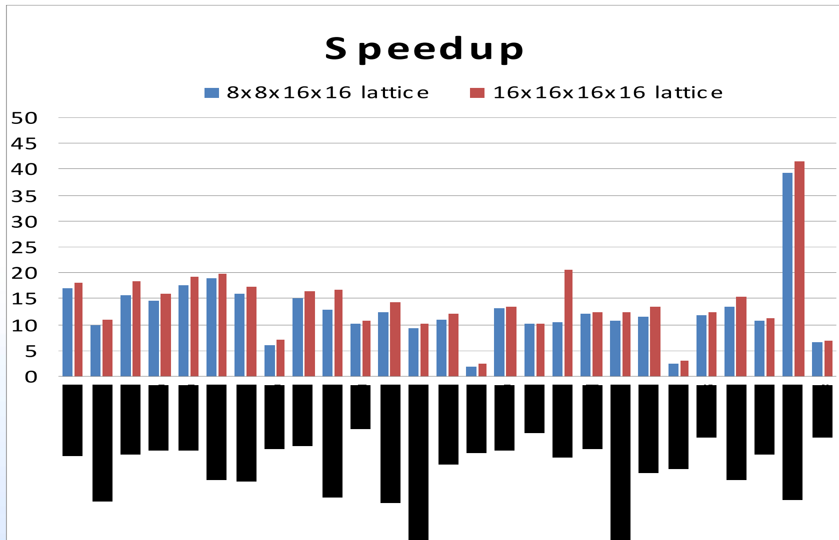


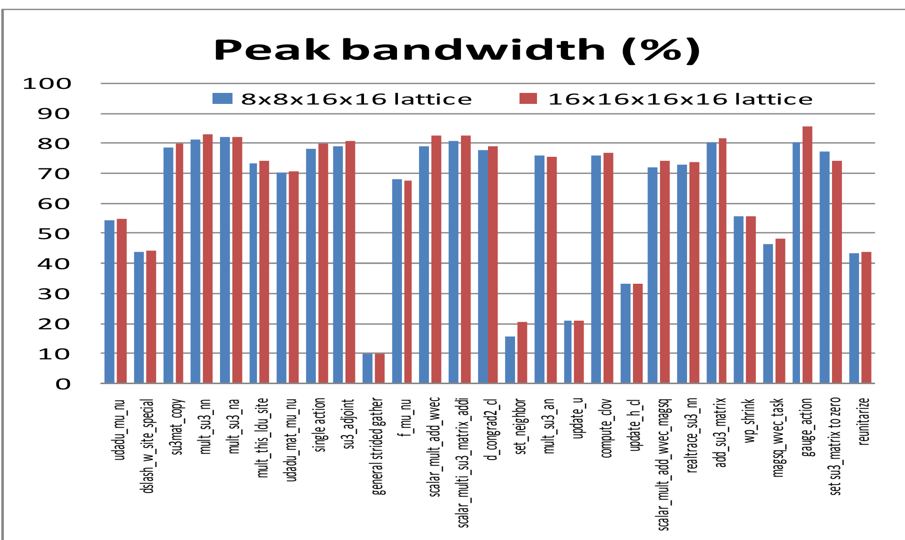
NCSA

MILC Kernel performance

- GFLOPS are low for all kernels
- Bandwidth is around 80% of peak for most of kernels
- Kernel speedup compared to CPU for most of kernels are between 10x to 20x
- set_memory_to_zero kernel has ~40x speedup
- Memcpy speedup >15x







NCSA

MILC Application performance

- Single Cell Application performance speedup
  - ~8–10x, compared to Xeon single core

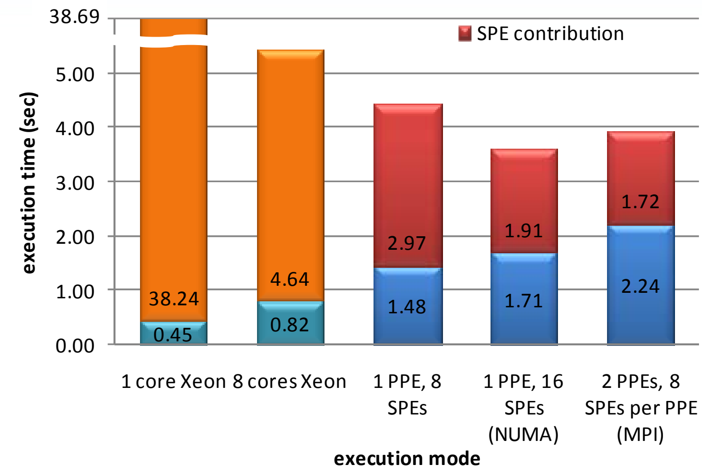
- Profile in Xeon
  - 98.8% parallel code, 1.2% serial code

    ⇩ speedup ⇩ slowdown

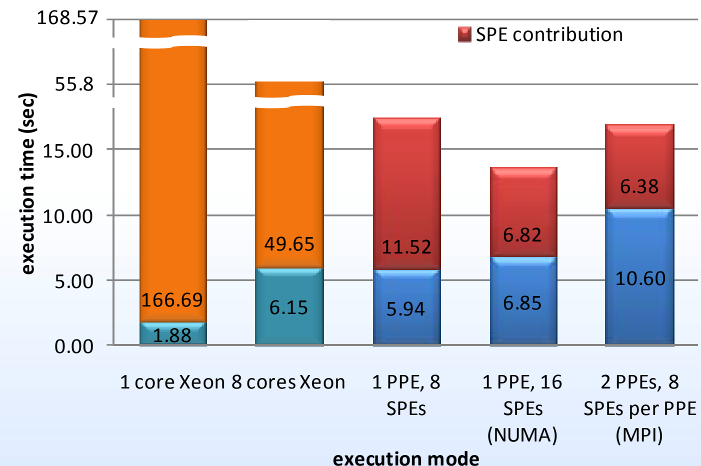
  - 67-38% kernel SPU time, 33-62% PPU time of overall runtime in Cell

➔PPE is standing in the way for further improvement

8x8x16x16 lattice



16x16x16x16 lattice



NCSA

Introduction –Quantum Chemistry

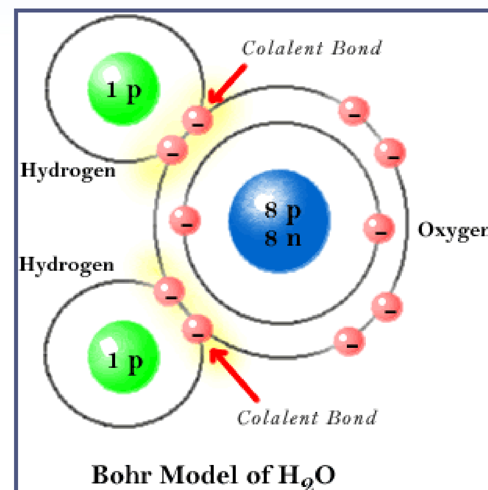
- Two basic questions in Chemistry* :
  1. Where are the electrons?
  2. Where are the nuclei?

  ➔Quantum Chemistry focuses the first question by solving the time-independent Schrödinger equation to get the electronic wave functions.

  And the absolute squire is interpreted as the probability distribution for the positions of the electrons.

- The probability distribution function is usually expanded to Gaussian type basis functions.

- To find the coefficients in the above expansion, we need do lots of two electron repulsion integrals



Bohr Model of H_2O

$$\chi_\mu(\vec{r}) \propto (x-x_\mu)^l (y-y_\mu)^m (z-z_\mu)^n \exp\left(-\alpha_\mu\left|\vec{r}-\vec{R}_\mu\right|^2\right)$$

Gaussian type basis functions

$$(\mu\nu|\lambda\sigma) = \iint \frac{\chi_\mu(\vec{r}_1)\chi_\nu(\vec{r}_1)\chi_\lambda(\vec{r}_2)\chi_\sigma(\vec{r}_2)}{\left|\vec{r}_1-\vec{r}_2\right|} d\vec{r}_1 d\vec{r}_2$$

two Electron Repulsion Integral (ERI)

* http://mtzweb.scs.uiuc.edu/research/gpu/

Introduction – Electron Repulsion Integral (ERI)

Reference CPU implementation

```
for (s1 = startShell; s1 < stopShell; s1++)
 for (s2 = s1; s2 < totNumShells; s2++)
  for(s3 = s1; s3 < totNumShells; s3++)
   for(s4=s3; s4 < totNumshells; s4++)

    for (p1=0;p1< numPrimitives[s1]; p1++)
     for (p2=0;p2< numPrimitives[s2]; p2++)
      for (p3=0;p3< numPrimitives[s3]; p3++)
       for (p4=0;p4< numPrimitives[s4]; p4++)
       {
             …..
        H_ReductionSum[s1,s2,s3,s4 += sqrt(F_PI*rho)*
             I1*I2*I3*Weight*Coeff1*Coeff2*Coeff3*Coeff4;
       }
```

- Four outer loops sequence through all unique combinations of electron shells
- Four Inner loops sequence through all shell primitives
- The primitives [ss|ss] are computed and summed up in the core code.

$$(\mu\nu|\lambda\sigma) = \sum_{p=1}^{N_\mu}\sum_{q=1}^{N_\nu}\sum_{r=1}^{N_\lambda}\sum_{s=1}^{N_\sigma} d_{\mu p}\, d_{\nu q}\, d_{\lambda r}\, d_{\sigma s}\,[pq|rs]$$

The general form for [ss|ss] integral

$$[s_1 s_2|s_3 s_4] = \frac{\pi^3}{AB\sqrt{A+B}} K_{12}(\vec{\mathbf{R}}_{12}) K_{34}(\vec{\mathbf{R}}_{34}) F_0\left(\frac{AB}{A+B}[\vec{\mathbf{R}}_{\mathrm{P}} - \vec{\mathbf{R}}_{\mathrm{Q}}]^2\right)$$

where

$$A = \alpha_1 + \alpha_2,\ B = \alpha_3 + \alpha_4,\ F_0(t) = \frac{erf(\sqrt{t})}{\sqrt{t}},$$

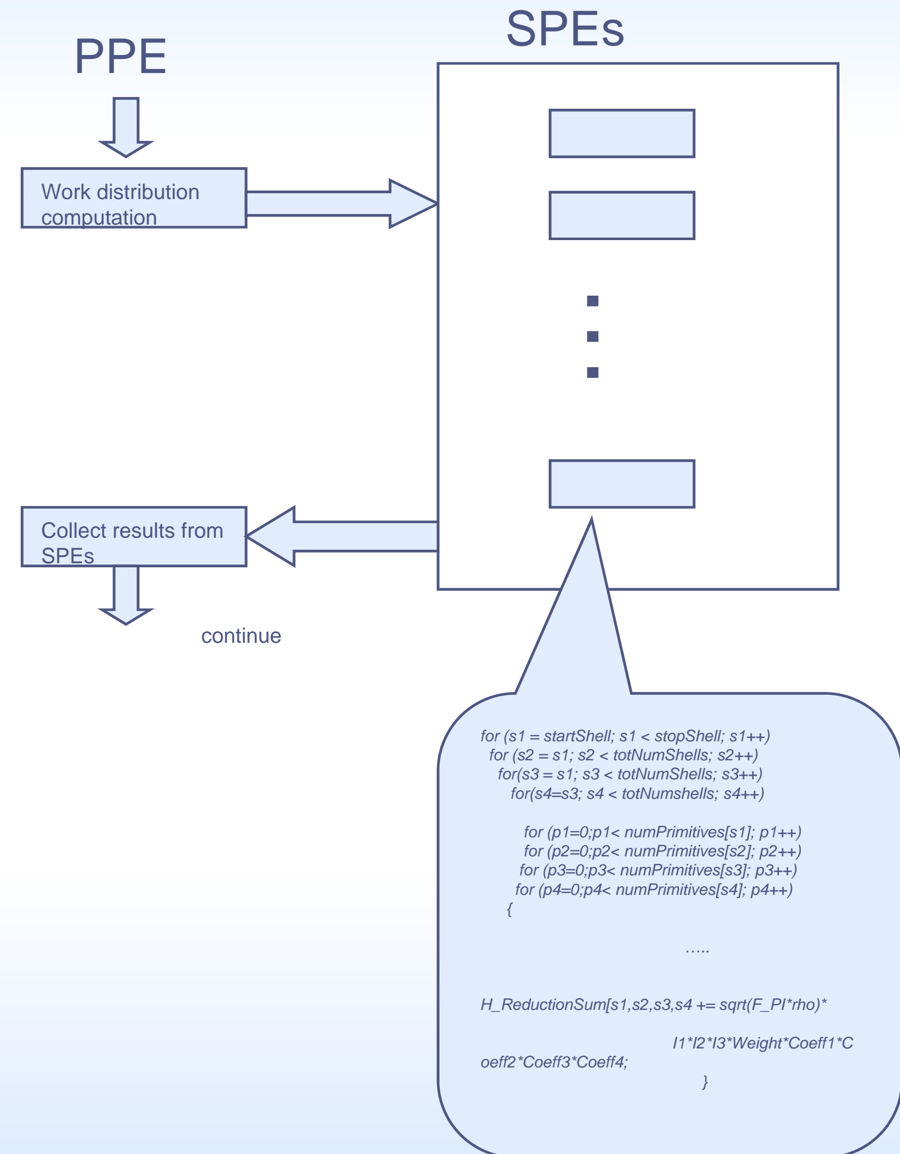
$$\vec{\mathbf{R}}_{\mathbf{kl}} = \vec{\mathbf{R}}_{\mathbf{k}} - \vec{\mathbf{R}}_{\mathbf{l}},\ \vec{\mathbf{R}}_{\mathbf{P}} = \frac{\alpha_1\vec{\mathbf{R}}_1 + \alpha_2\vec{\mathbf{R}}_2}{A},\ \vec{\mathbf{R}}_{\mathbf{Q}} = \frac{\alpha_3\vec{\mathbf{R}}_3 + \alpha_4\vec{\mathbf{R}}_4}{B},$$

$$K_{ij}(\vec{\mathbf{R}}_{ij}) = exp\left(-\frac{\alpha_i\alpha_j}{\alpha_i + \alpha_j}[\vec{\mathbf{R}}_i - \vec{\mathbf{R}}_j]^2\right)$$

NCSA

Porting to Cell B.E. – load balance

- Function offload programming
- Each SPE is assigned a range of (s1,s2,s3,s4) to work on.
- Load balance Computation in PPE
  - Each primitive integral roughly has the same amount of computation
  - Each contracted integral may have different amount of computation
  - Each iteration in outer most loop has different amount of computation
  - PPE must compute the amount of computation before SPEs run



NCSA

SPE kernels

- Is local store enough:
  - Input data: an array of coordinates + an array of shells + an array of primitives < 32 KB
  - The Gauss error function -- erf() – is evaluated by interpolating table. The table size is < 85KB
  - We have enough local store
- Precomputing is necessary to reduce redundant computation
  - Precomputed intermediate results are much larger than local store

NCSA

SPE kernel -- precompute

- Instead of computing every primitive integrals from equations, we precompute pairwise quantities and store them in main memory
- DMA all pairwise quantities before computing a contracted integral
- Precomputed quantities:

$$\alpha_1 + \alpha_2$$

$$d_1 d_2 \left(\frac{\pi}{\alpha_1 + \alpha_2}\right)^{3/2} e^{-\alpha_1\alpha_2/(\alpha_1+\alpha_2)\vec{R}_{12}^{\,2}}$$

$$(\alpha_1 \vec{R}_1 + \alpha_2 \vec{R}_2)/(\alpha_1 + \alpha_2)$$

- Trade bandwidth with computation

```
for (s1 = startShell; s1 < stopShell; s1++)
  for (s2 = s1; s2 < totNumShells; s2++)
    for(s3 = s1; s3 < totNumShells; s3++)
      for(s4=s3; s4 < totNumshells; s4++)

        DMA in precomputed pair quantities.

        for (p1=0;p1< numPrimitives[s1]; p1++)
         for (p2=0;p2< numPrimitives[s2]; p2++)
          for (p3=0;p3< numPrimitives[s3]; p3++)
           for (p4=0;p4< numPrimitives[s4]; p4++)
           {
                   …..
            H_ReductionSum[s1,s2,s3,s4 += sqrt(F_PI*rho)*
                   I1*I2*I3*Weight*Coeff1*Coeff2*Coeff3*Coeff4;
           }
```

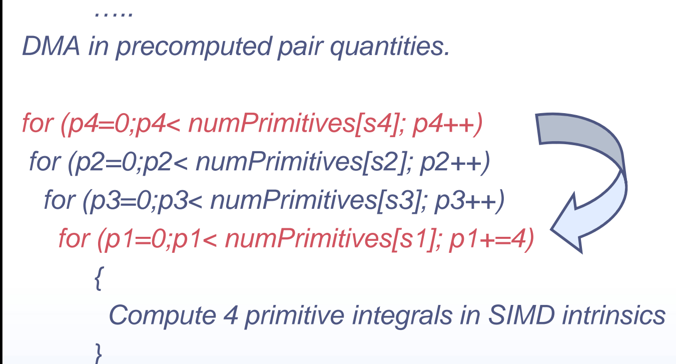

NCSA

SPE kernel – inner loops optimizations

- Naïve way of SIMDing kernel
  - Use a counter to keep track of the number of primitive integrals
  - If counter >=4, do a 4-way computation
- Loop switch
  - If one of the loops' length is multiple of 4, we switch the loop to the inner most
  - Advance in increment of 4, and get rid of counter
- Loop unrolling
  - If the numPrimitives are the same for all primitive integral and the happened to be some nice number
  - Completely unroll the inner loops generate the most efficient code

```
for (p1=0;p1< numPrimitives[s1]; p1++)
 for (p2=0;p2< numPrimitives[s2]; p2++)
  for (p3=0;p3< numPrimitives[s3]; p3++)
   for (p4=0;p4< numPrimitives[s4]; p4++)
   {    static int count =0;
        count++;
        if (count == 4){
          compute 4 primitive integrals using SIMD intrinsics
        }
   }
```

Naïve implementation of SIMD kernel



Loop switch: loop1 length is multiples of 4, switch loop1 and loop4

NCSA

Performance results

	Model1	Model2
# of Atoms	30	64
Basis set	6-311G	STO-6G
# of integrals	528,569,315	2,861,464,320
# of reduction elements	3,146,010	2,207,920
Xeon (2.33Ghz)	21.04	112.55
Cell-blade (16 SPEs)	1.1	0.92
Speedup	19x	122x.

- Model1 is 10 water molecules (30 atoms in total), model2 is 64 hydrogen atom arranged in lattice
- Model1 has non-uniform contracted integral intensity, ranging from 4096 to just 1 primitive integrals.
  - Loop switching is not always possible ➔ naïve SIMD implementation ➔ more control overhead
  - Loop unrolling is not possible since the # of iterations in each inner loop changes
  - Precomputing proves to slow down due to DMA overhead
- Model2 uniform computation intensity
  - Precomputing and loop unrolling proves to be efficient
  - Module outperforms GPU implementation*

** Quantum Chemistry on Graphical Processing Units. 1. Strategies for Two-Electron Integral Evaluation. Ivan S. Ufimtsev and Todd J. Martinez, Journal of Chemical Theory and Computation, February 2008*

Summary

	Ported code	Precision	Performance Limit factor	Cell Blade to single core speedup[1]	Cell blade to Xeon blade speedup[2]
NAMD	Modified kernel	SP,DP	Computation	22-26x	NULL
MILC	application	SP	Bandwidth	10-12x	1.5-4.1x
ERI	Kernel	SP	Computation	19-122x	NULL

1. *CPU comparing core for NAMD 3.0Ghz Xeon, for MILC and ERI it is 2.33 Ghz Xeon*
2. *One Xeon blade contains 8 cores with 2MB L2 cache per core.*

Lessons learned

- Keep code on PPE to a minimum
  - 1.2% runtime code ➔ 33-62% in PPE in MILC
- Find out the limiting factor in the kernel and work on it
  - MILC is bandwidth limited and we focus on improving the usable bandwidth
  - NAMD and ERI is compute-bound and we focus on optimizing the SIMD kernel.
- Sometimes we can trade bandwidth with computation or vice versa
  - In ERI, depends on input data, we can either precompute some quantities in PPE and DMA in or do all computation in SPEs
- Application data layout in memory in critical
  - Padding would not be necessary if MILC is field centered ➔improvement of performance and productivity
  - Proper data layout makes SIMDizing kernel code easier and make it faster

Thank You

- Questions?

NCSA