QCD on GPUs

Pushan Majumdar

(Department of Theoretical Physics, IACS, Kolkata)

Perspectives and challenges in lattice gauge theory

TIFR 2015, Mumbai

Outline

- Introduction
- OpenACC standard & code samples
- Performance : Speed-up
- Observations
- Wishlist for GPU computing

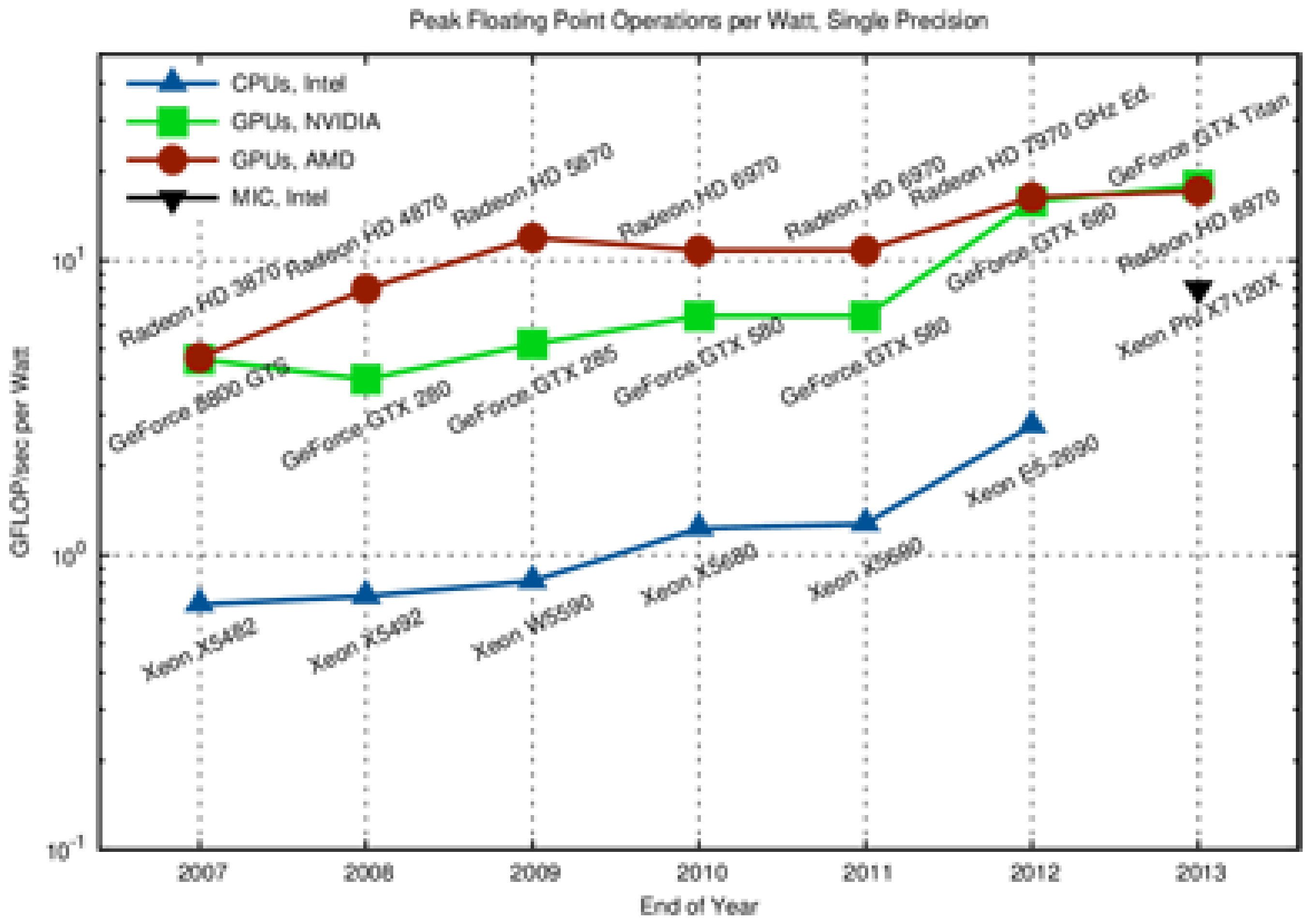
Introduction

- With single core CPU performance being stagnant, efficient parallelization has become important for lattice QCD computations. Fortunately lattice QCD codes can be parallelized relatively easily.

- Parallelization between different nodes using MPI like platforms have been around for a long time. New thing is parallelization among large number of cores on the same chip.

- Architectures under consideration are
  Multicore CPUs $\sim$ 15 cores on a chip $\sim$ 2GB / core
  Xeon-PHIs $\sim$ 60 cores on a chip $\sim$ 250 MB / core
  GPUs $\sim$ 500 or 1000 cores on a chip $\sim$ 12 MB / core

- Same OpenMP code can run on multi-core CPUs and Xeon-PHIs in the native mode. GPUs require more effort.


Peak Floating Point Operations per Watt, Single Precision
CPUs, Intel
GPUs, NVIDIA
GPUs, AMD
MIC, Intel
GFLOP/sec per Watt
End of Year
2007
2008
2009
2010
2011
2012
2013
10^{-1}
10^{0}
10^{1}
Radeon HD 3870
Radeon HD 4870
Radeon HD 5870
Radeon HD 6970
Radeon HD 6970
Radeon HD 7970 GHz Ed.
Radeon HD 8970
GeForce 8800 GTS
GeForce GTX 280
GeForce GTX 285
GeForce GTX 580
GeForce GTX 580
GeForce GTX 680
GeForce GTX Titan
Xeon Phi X7120X
Xeon X5482
Xeon X5492
Xeon W5590
Xeon X5680
Xeon X5690
Xeon E5-2690

- The first attempts to run lattice simulations on GPUs was around 2005 :

  Lattice QCD as a video game

  G. I. Egri, Z. Fodor, C. Hoelbling, S. D. Katz, D. Nogradi, K. K. Szabo

  Comput.Phys.Commun.177:631-639,2007

- A more systematic resource is maintained by M. A. Clark et. al. at `https://github.com/lattice/quda` :

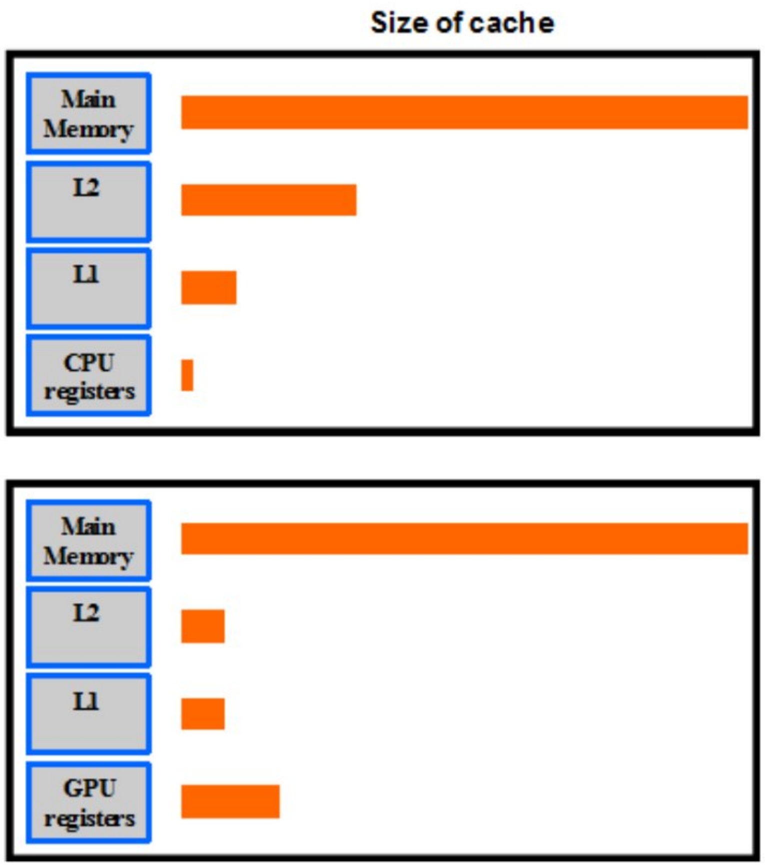
  Solving Lattice QCD systems of equations using mixed precision solvers on GPUs

  M. A. Clark, R. Babich, K. Barros, R. Brower, and C. Rebbi

  Comput. Phys. Commun. 181, 1517 (2010) [arXiv:0911.3191 [hep-lat]].

- Later pure gauge theory codes were also ported to GPUs by Bicudo et. al.

Memory considerations for GPUs



Register memory is larger than cache in GPUs.

CUDA puts arrays into the main memory while scalar variables are put in registers.

Code generators can automatically convert arrays into lots of scalar variables.

This can increase the performance of a CUDA code by a several factors.

- GPUs are programmed primarily with CUDA which is a C like language.
- It takes a reasonable effort to port existing codes to CUDA and run them efficiently.
- A lot of existing lattice QCD codes are written in FORTRAN.
- CUDA FORTRAN exists but is not very useful. It only uses wrappers around the CUDA C functions for interfacing with FORTRAN routines.
- For smaller groups a much more viable option is to use directives based programming such as OpenACC.

The OpenACC standard

OpenACC is a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems. *from Wikipedia*

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators. *from openacc.org*

The directives and programming model defined in this document allow programmers to create high-level host+accelerator programs without the need to explicitly initialize the accelerator, manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown.

from openacc.org

Programs I have had some experience with :

1. Staggered fermions with wilson gauge action on
   (a) single GPU – in some detail
   (b) multi GPU – preliminary

2. Wilson fermions with Wilson gauge action on
   single GPU – preliminary

Main bottlenecks is slow data movement between CPU & GPU. Speed is about 5 GB/s or 8 GB/s depending on whether it PCI 2.0 or 3.0.

Impossible to avoid CPU completely as I/O , if-then clause is evaluated on CPU.
BLAS functions are launched from CPU and MPI calls (at least for Fermi GPUs) are launched from CPUs.

Hybrid Monte Carlo - Parallelization Considerations

- Langevin step: Requires between 10^6 and 10^{10} random numbers at each step. Still generating on CPU and copying to GPU.

- Molecular dynamics step: Involves a conjugate gradient. Expensive. Takes up 85-90% of simulation time. Target for parallelization via OpenACC.

- Metropolis Accept/Reject step : Intrinsically serial step.

- The basic matrix vector operation requires 66 Flops and 120 bytes of data movement. Thus lattice QCD computations are bandwidth limited.

- CPU to main memory bandwidth is around 25 GB/s, the maximum performance without reusing data is about 13 GFlops. A similar calculation for the GPU would yield figures around 90 GFlops for a X2090 or 144 GFlops for a K40M.

- The main challenge in GPU programming for lattice QCD codes is how to get around these bottlenecks.

- To get any reasonable speed-up, at least the entire conjugate gradient must be on the GPU. Data must be copied from CPU to GPU at the start of the routine and the result copied back at the end of the routine.

Single GPU code

```
subroutine congrad(nitcg)
```

. . . *All kinds of definitions and declarations* . . .

```
!$ACC  data copy(nitcg,alpha,betad,betan)
!$ACC+ copyin(nx,iup,idn,u,r)
!$ACC+ copyout(x,y)
!$ACC+ create(ud,ap,atap,p)
*
      call linkc_acc
!!$OMP parallel do default(shared)
!$ACC  parallel loop collapse(2) reduction(+:betan) present(p,r,x)
      do l = 1, mvd2
      do ic=1,nc
         p(l,ic) = r(l,ic) ; x(l,ic) = (0.,0.)
         betan=betan+conjg(r(l,ic))*r(l,ic)
      end do
      end do
```

```
!       betan=real(zdotc(mv3d2,r,1,r,1))
!$ACC  update host(betan)
      if (betan.lt.delit) go to 30
!$ACC  parallel present(beta,betan,betad,alphan)
       beta=betan/betad ; betad=betan ; alphan=betan
!$ACC  end parallel
```

`do nx = 1, nitrc` *Main loop of conjugate gradient begins*

nitcg $\leftarrow$ nitcg+1   ;   ap = 0

call fmv(0,mvd2,ap,p) $\rightarrow$*(Matrix-vector multiplication)*

alphad=$\langle$ap,ap$\rangle$ + $\langle$p,p$\rangle$;   alpha=alphan/alphad

atap $\leftarrow$ p   ;   x $\leftarrow$ x + alpha $*$ p

```
       call fmtv(atap,ap)    →(Matrix-vector multiplication)

       r ← r - alpha * atap
       betan=⟨r, r⟩
!$ACC  update host(betan)         Exit condition evaluated on CPU
       if (betan .lt. delit) go to 30
       beta=betan/betad ; betad=betan ; alphan=betan
       p ← r + beta * p
       end do        Main loop of conjugate gradient ends
30     continue
*
       y = 0         Solution on the second half lattice
       call fmv(mvd2,mv,y,x)    →(Matrix-vector multiplication)
*
!$ACC  end data
       return
```

Matrix-vector multiplication routine

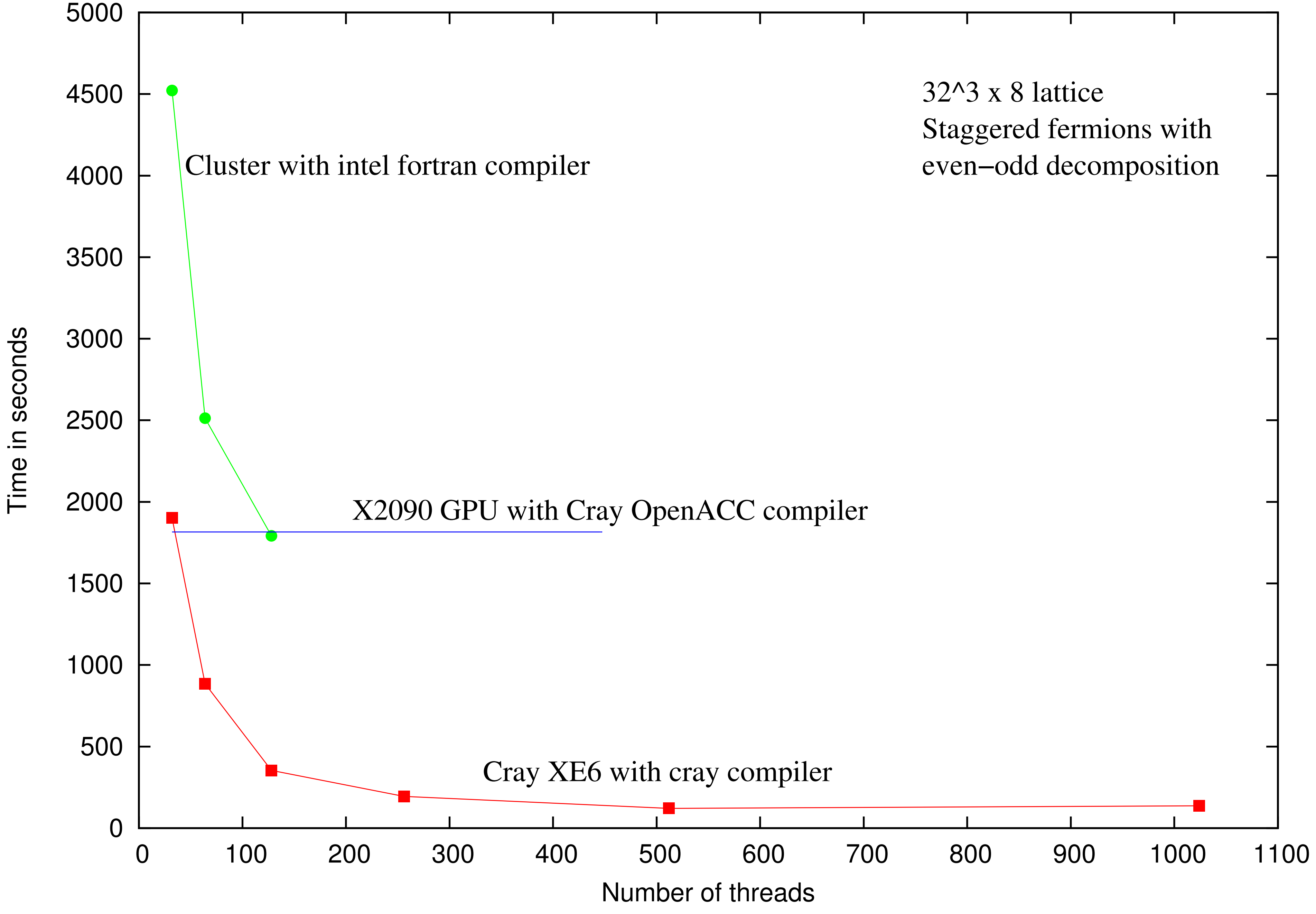
```
      subroutine fmv(noff,nsz,v,w)
```

. . . *All kinds of definitions and declarations* . . .

```
!!$OMP parallel do default(shared)
!!$OMP+      private(nnu,px1,px2,px3,px4,px5,px6)
!!$OMP^ private(v1,v2,v3)
!$ACC  parallel loop present(u,ud,v,w,iup,idn)
!$ACC+ private(nnu,px1,px2,px3,px4,px5,px6,v1,v2,v3)
!$ACC+ vector_length(32)
      do  l = noff+1, noff+mvd2
        ⋮
```

Routine identical to CPU version

```
        ⋮
      enddo
      return
```



32^3 x 8 lattice
Staggered fermions with
even−odd decomposition
Cluster with intel fortran compiler
X2090 GPU with Cray OpenACC compiler
Cray XE6 with cray compiler
Time in seconds
Number of threads
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
0
100
200
300
400
500
600
700
800
900
1000
1100

Multi-GPU code

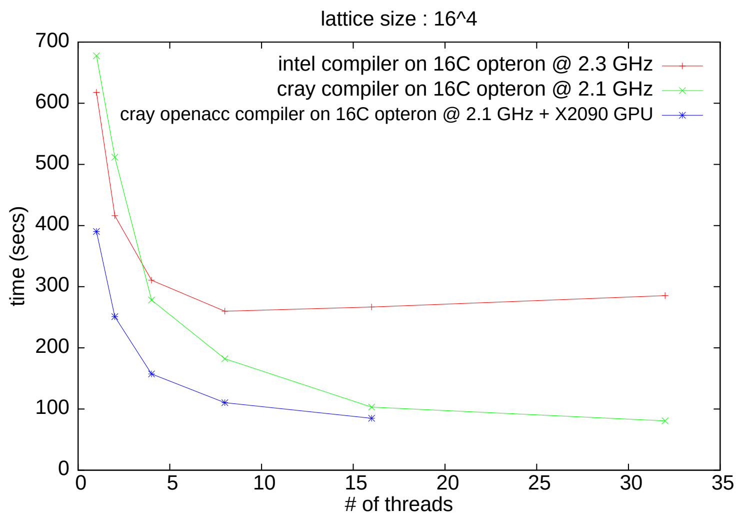
```
         ⋮
          ap_loc ← 0
!$ACC  parallel loop present(u,ud,ap_loc,p,iup,idn)
      do  l = base+1, base+nvd2
        v1 = ap_loc(1,l-base)
         ⋮
```

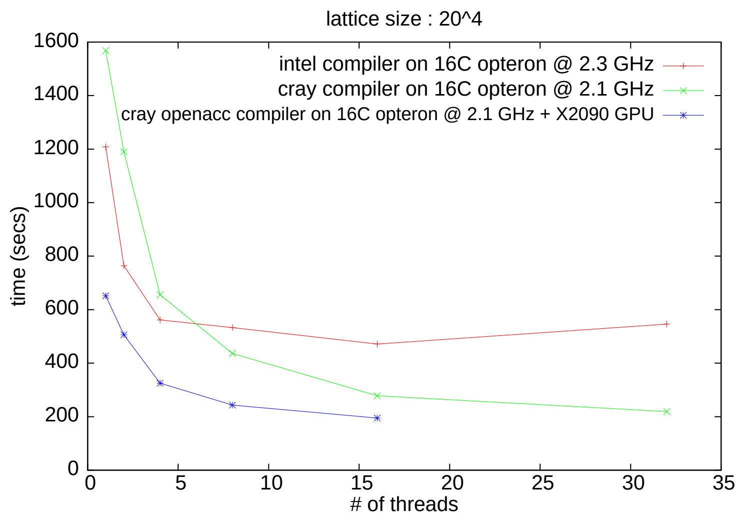
Lines identical to scalar version

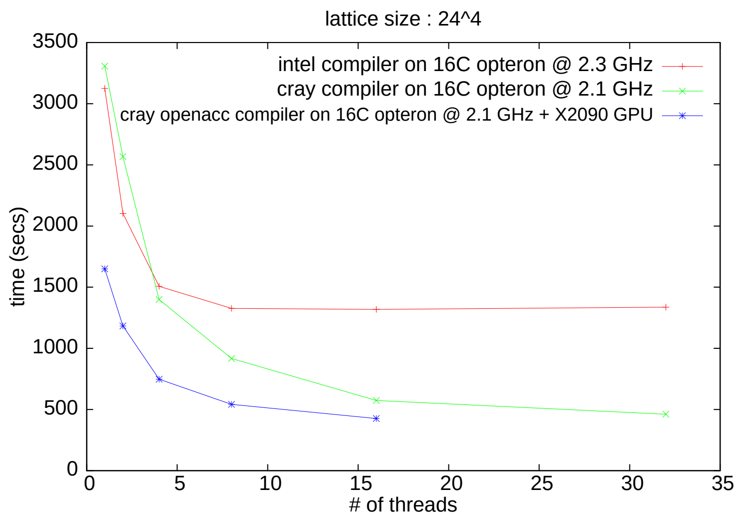
```
         ⋮
        ap_loc(3,l-base) = v3
      enddo
!$ACC update host(ap_loc)
      call MPI_ALLGATHER(ap_loc,3*nvd2,MPI_DOUBLE_COMPLEX,
     +          ap,3*nvd2,MPI_DOUBLE_COMPLEX,MPI_COMM_WORLD,ierr)
!$ACC  update device (ap)
```

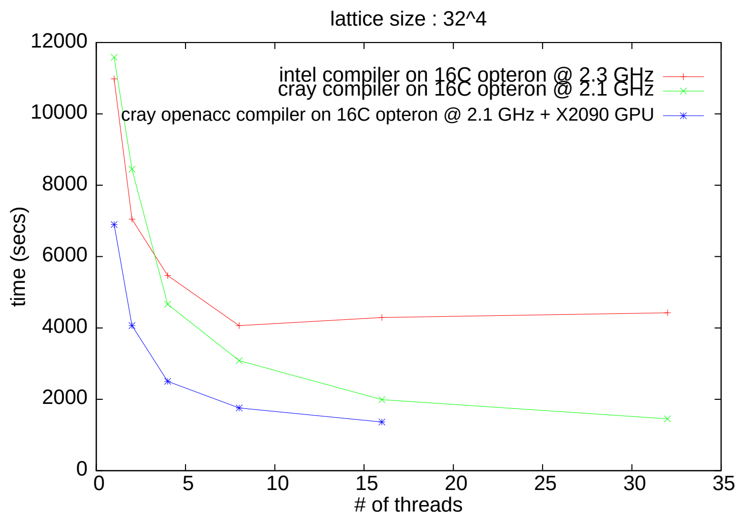
Worry about async compiler options.

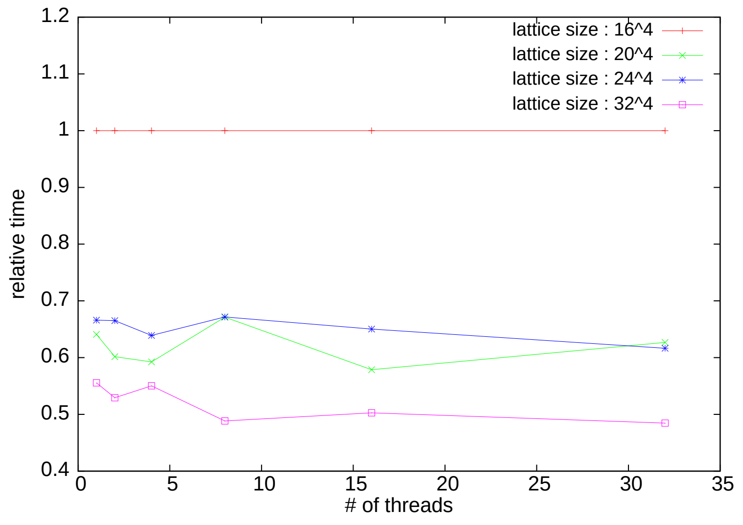
Performance comparison intel vs cray vs openacc

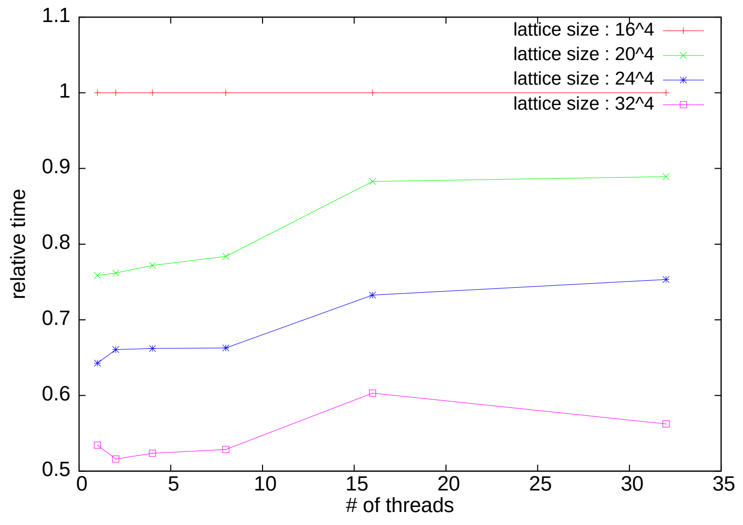


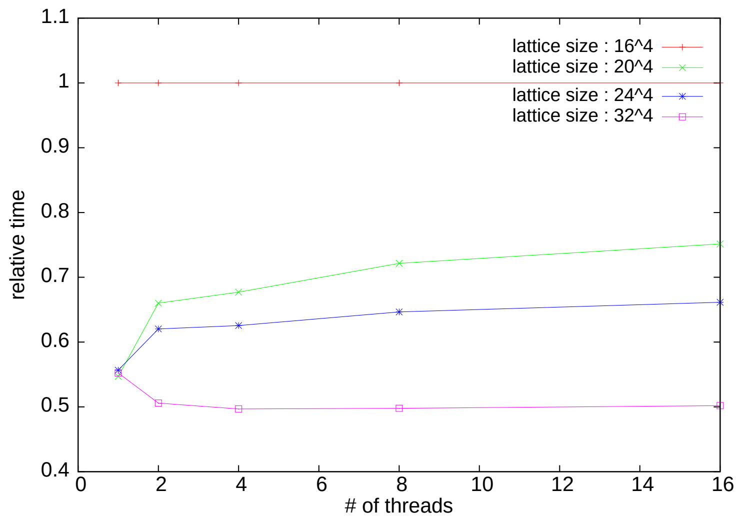












Performance of the different compilers as data size increases.

Timings have been scaled using the scaling of HMC :

time $\propto$ (vector size)$^{5/4}$.

Larger data is beneficial.

Top left : intel compiler;

Top right : cray compiler;

Bottom : cray openacc compiler

# of threads	wrong order	right order
1	673.6	651.69
2	623.1	505.92
4	522.7	325.22
8	474.5	243.2
16	447.7	194.7

Global memory access in GPU is slow. Therefore correct ordering of array index is essential for arrays being declared on the GPU. There is a 3.5 times difference in speed of the conjugate gradient depending how the array is defined.

array(3,4,nsite) or array(nsite,3,4) nsite $\sim$ 160000.

The first is faster on the CPU while the second is faster on the GPU.

While the conjugate gradient speeded up another routine got slowed down by a factor 2. Try to put most of the program on the GPU or maintain copies of arrays and synchronize them.

Observations

- GPUs provide affordable (in terms of both money and power) computing resources.

- OpenACC : Easy path to go from the CPU to the GPU.

- Coding effort is only marginally higher than OpenMP. Almost each OpenMP directive can be replaced with a OpenACC directive. Only additional directive is the creation of a data region.

- Real gain comes only when a large chunk of the program such as the whole conjugate gradient routine is on the GPU.

- Performance of single GPU is roughly equivalent to 128 cores of cluster with QDR infiniband interconnect.
- For the programs analyzed, the advantage of the GPU increases with increasing data size.
- There can be a huge penalty in performance (several times) for data access on the GPU if the data is not optimally organized.
- Comparing between an optimized pure CPU MPI code and an OpenACC code on a single Cray node, the MPI code is faster for lattice sizes of about 20^4. Beyond that the OpenACC code is faster.

- Further performance gains can be obtained by using mixed-precision routines and improved storage schemes.

- PGI compiler does not have support for double precision complex multiplication support in its OpenACC compiler. Implementation in next release.

Wishlist for the future

- For data already on GPU, launch BLAS & LAPACK routines from GPU.

- For subroutine calls from within an OpenACC region, local variables defined in the subroutines have to be copied to the GPU at the beginning of the OpenACC region. This is a bit awkward.

- Currently the fortran functions dot_product and matmul are not supported on the GPU. Add support for that.

- Support for array handling features of Fortran90.

- GCC 5 has introduced support for openACC but there are issues still.

- IBM working in conjunction with NVIDIA to remove the PCI bottle-neck. Accelerator to connect to high bandwidth bus (like QPI or Hypertransport) - NVLINK.

- Mellanox has come out with a 100 Gbps Infiniband switch.

Acknowlegdments

ILGTI-TIFR for funding the GPU portion of the Cray on which these studies were carried out. IACS for funding the rest of the Cray without which the GPU portion wouldn't run.

Cray India team for help at various stages during the development of the OpenACC codes.

Thank You