Tuning a C++ Application for the Latest Generation x64 Processors with PGI Compilers and Tools

Doug Doerfler – dwdoerf@sandia.gov
David Hensigner – dmhensi@sandia.gov
Brent Leback – brent.leback@pgroup.com
Doug Miles – douglas.miles@pgroup.com

CUG Seattle
May, 2007

Introduction

- **What we did last Summer** – project overview
- **SSE vectorization on x64 CPUs –** new and improved!
- **Performance characteristics of x64 processors**
- **Optimizing the cache oblivious C++ ALEGRA kernel**
- **Conclusions and Future Work**

Project Background

- To obtain a highly optimized code …
  - Requires expert knowledge at all levels, from algorithms, to applications, to compilers, to processor architecture
  - Although “-fast” is usually pretty good, if you’re serious you need to engage compiler expertise
- Sandia National Labs and PGI have been collaborating since early days of ASCI Red
- Here’s the latest collaborative effort …

Optimizing a 2D Lagrangian Hydro Code

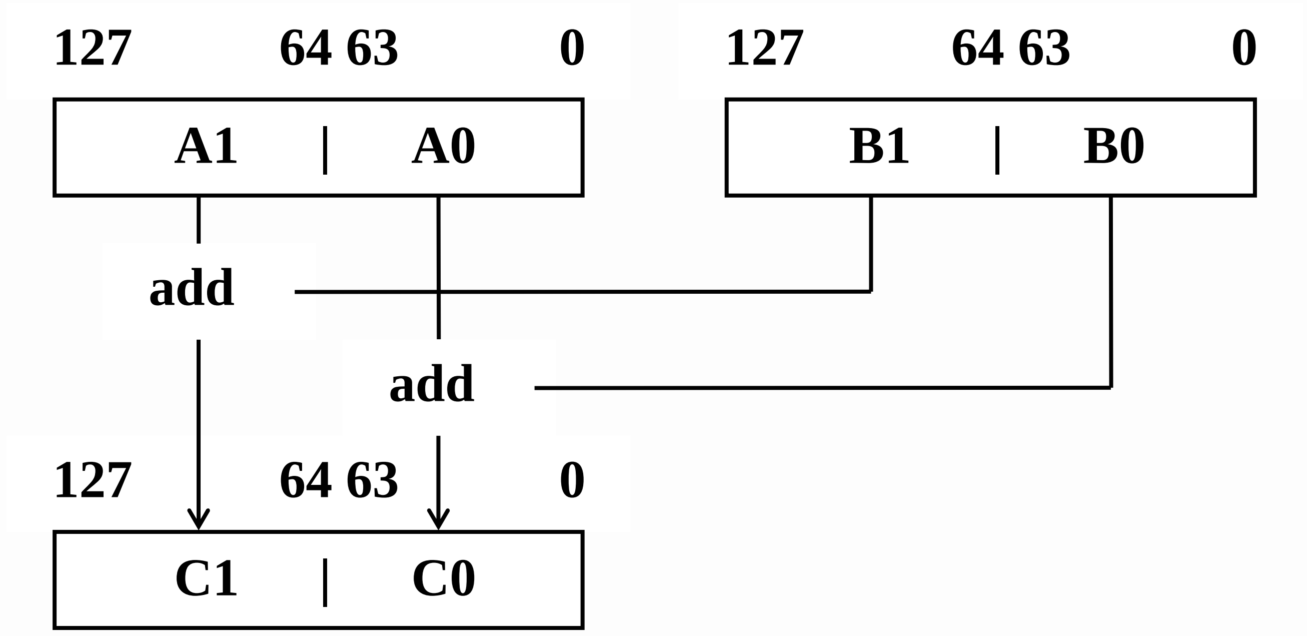
- A cache oblivious implementation by Hensinger, Frigo and Strumpen, CUG 2006
- Recursively walks through multiple time steps over subsets of the spatial data domain

  … which reduces memory bandwidth requirements

  … and hence is a good candidate for vector optimizations
- Originally coded as an array of structures
- Application level optimization …
  - Original data is organized as an array of structures (acceleration, velocity and force per structure element)
  - Reorder data layout to a structure of arrays

    … which allows effective loading of elements into vector registers
  - Did not change cache oblivious techniques
- Compiler level optimization …

Double-precision Packed SSE Operations on x64 CPUs



x64 Double-precision Packed SSE Implementations

1st Gen

```
Cycle i:        [A1|A0]   [B1|B0]
                    \         /
                     A0+B0
                     .....
                     .....
Cycle i + 1:   [A1|A0]   [B1|B0]
                    \         /
                     A1+B1
                     A0+A0
                     .....
Cycle i + p - 1:   .....
                    A1+B1
                    A0+B0
                        \
                    [ .. |C0]
Cycle I + p:       .....
                   .....
                   A1+B1
                     /
                  [C1|C0]
```

2nd Gen

```
Cycle i:      [A1|A0]   [B1|B0]
                  \         /
              [A1+B1|A0+B0]
              .....
              .....
Cycle i + 1: .....
              [A1+B1|A0+A0]
              .....
Cycle i + p: .....
              .....
              [A1+B1|A0+B0]
                  \       /
                  [C1|C0]
```

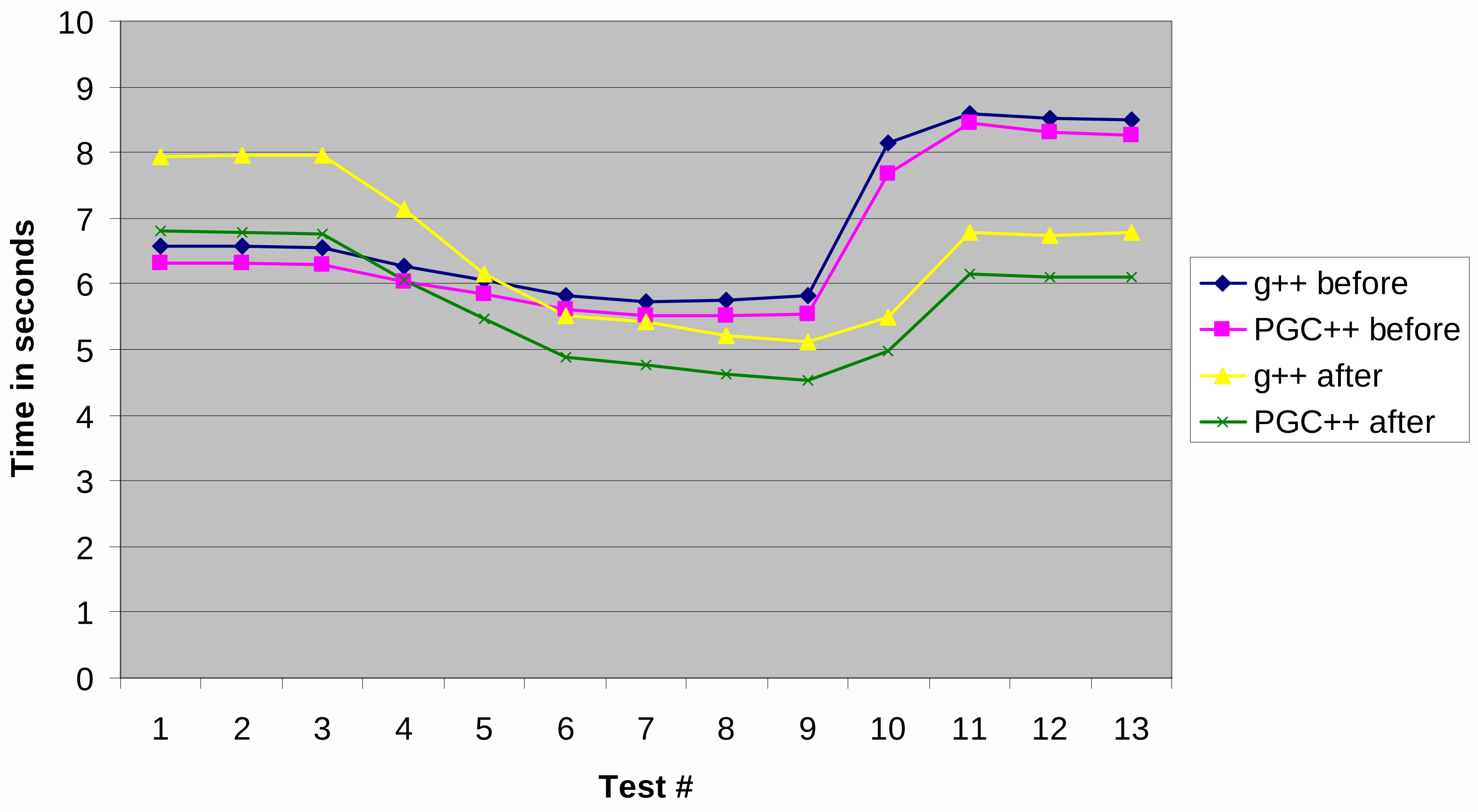
Break Out to Assembly Code Kernels View …

Percentage of Latest-generation x64 Peak Performance - Measured

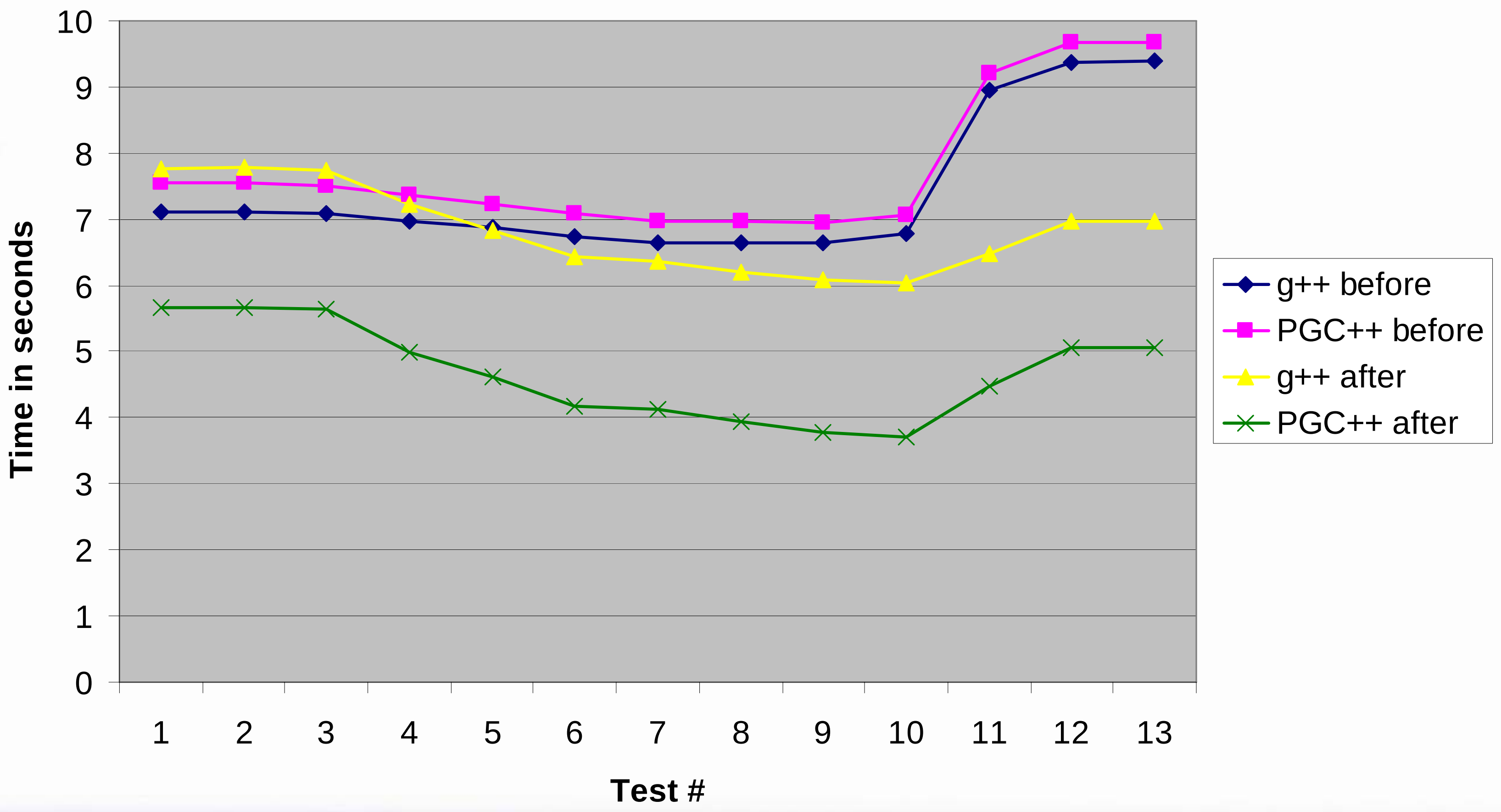
Memory Accesses per Mul-Add	*AMD*		*Intel*	
	First-Gen AMD64	Latest-Gen AMDOpteron	First-Gen EM64T	Latest-Gen Intel Core 2
0 Bytes Register-to-Register Scalar	50%	50%	25%	50%
0 Bytes Register-to-Register Vector	50%	100%	50%	100%
8 Bytes Aligned	50%	100%	50%	100%
8 Bytes Unaligned	25-48%	50-90%	28%	38-40%
16 Bytes Aligned	47%	90%	38%	50%
16 Bytes Unaligned	22-25%	25-65%	17-20%	25%
24 Bytes Aligned	33%	65%	27%	33%
24 Bytes Unaligned	13-22%	17-65%	12-20%	17-25%
32 Bytes Aligned	24%	50%	20%	25%
32 Bytes Unaligned	10-14%	13-50%	10-13%	12-17%

Break Out to Source Code View …

1st Generation x64 - AMD Opteron Alegra Kernel Performance



2nd Generation x64 - Intel Core 2 Alegra Kernel Performance



Conclusions

- Significant performance gains possible by writing C/C++ codes to maximize vectorization
- Similar improvements can potentially be applied to ALEGRA, which accounted for 4.5% of all DoD HPCMP computing cycles in 2006
- Communication and close cooperation between end-users and compiler writers can pay large dividends

Q & A?