Compilers

PGI

- Recommended first compile/run
  - -fastsse –tp barcelona-64
- Get diagnostics
  - -Minfo –Mneginfo
- Inlining
  - **–Mipa=fast,inline**
- Recognize OpenMP directives
  - -mp=nonuma
- **Automatic parallelization**
  - **-Mconcur**

Pathscale

- Recommended first compile/run
  - Ftn –O3 –OPT:Ofast -march=barcelona
- Get Diagnostics
  - -LNO:simd_verbose=ON
- Inlining
  - -ipa
- Recognize OpenMP directives
  - -mp
- **Automatic parallelization**
  - **-apo**

PGI Basic Compiler Usage

- A compiler driver interprets options and invokes pre-processors, compilers, assembler, linker, etc.
- Options precedence: if options conflict, last option on command line takes precedence
- Use -Minfo to see a listing of optimizations and transformations performed by the compiler
- Use -help to list all options or see details on how to use a given option, e.g. pgf90 -Mvect -help
- Use man pages for more details on options, e.g. "man pgf90"
- Use –v to see under the hood

Flags to support language dialects

- Fortran
  - pgf77, pgf90, pgf95, pghpf tools
  - Suffixes .f, .F, .for, .fpp, .f90, .F90, .f95, .F95, .hpf, .HPF
  - -Mextend, -Mfixed, -Mfreeform
  - Type size –i2, -i4, -i8, -r4, -r8, etc.
  - -Mcray, -Mbyteswapio, -Mupcase, -Mnomain, -Mrecursive, etc.
- C/C++
  - pgcc, pgCC, aka pgcpp
  - Suffixes .c, .C, .cc, .cpp, .i
  - -B, -c89, -c9x, -Xa, -Xc, -Xs, -Xt
  - -Msignextend, -Mfcon, -Msingle, -Muchar, -Mgccbugs

Specifying the target architecture

- Use the “tp” switch. Don’t need for Dual Core
  - -tp k8-64 or –tp p7-64 or –tp core2-64 for 64-bit code.
  - -tp amd64e for AMD opteron rev E or later
  - -tp x64 for unified binary
  - -tp k8-32, k7, p7, piv, piii, p6, p5, px for 32 bit code
  - -tp barcelona-64

Flags for debugging aids

- -g generates symbolic debug information used by a debugger
- -gopt generates debug information in the presence of optimization
- -Mbounds adds array bounds checking
- -v gives verbose output, useful for debugging system or build problems
- -Mlist will generate a listing
- -Minfo provides feedback on optimizations made by the compiler
- -S or –Mkeepasm to see the exact assembly generated

Basic optimization switches

- Traditional optimization controlled through -O[<n>], n is 0 to 4.
- -fast switch combines common set into one simple switch, is equal to -O2 -Munroll=c:1 -Mnoframe -Mlre
  - For -Munroll, c specifies completely unroll loops with this loop count or less
  - -Munroll=n:<m> says unroll other loops m times
- -Mlre is loop-carried redundancy elimination

Basic optimization switches, cont.

- fastsse switch is commonly used, extends –fast to SSE hardware, and vectorization
- -fastsse is equal to -O2 -Munroll=c:1 -Mnoframe -Mlre (-fast) plus -Mvect=sse, -Mscalarsse -Mcache_align, -Mflushz
- -Mcache_align aligns top level arrays and objects on cache-line boundaries
- -Mflushz flushes SSE denormal numbers to zero

Node level tuning

- **Vectorization** – packed SSE instructions maximize performance
- **Interprocedural Analysis (IPA)** – use it! motivating examples
- **Function Inlining –** especially important for C and C++
- **Parallelization** – for Cray multi-core processors
- **Miscellaneous Optimizations** – hit or miss, but worth a try

What can Interprocedural Analysis and Optimization with –Mipa do for You?

- Interprocedural constant propagation
- Pointer disambiguation
- Alignment detection, Alignment propagation
- Global variable mod/ref detection
- F90 shape propagation
- Function inlining
- IPA optimization of libraries, including inlining

Effect of IPA on the WUPWISE Benchmark

PGF95 Compiler Options	Execution Time in Seconds
–fastsse	156.49
–fastsse –Mipa=fast	121.65
–fastsse –Mipa=fast,inline	91.72

- –Mipa=fast => constant propagation => compiler sees complex matrices are all 4x3 => completely unrolls loops
- –Mipa=fast,inline => small matrix multiplies are all inlined

Using Interprocedural Analysis

- **Must be used at both compile time and link time**
- **Non-disruptive to development process – edit/build/run**
- **Speed-ups of 5% - 10% are common**
- **–Mipa=safe:<*name*> - safe to optimize functions which call or are called from unknown function/library *name***
- **–Mipa=libopt – perform IPA optimizations on libraries**
- **–Mipa=libinline – perform IPA inlining from libraries**

Explicit Function Inlining

–Minline[=[lib:]<inlib> | [name:]<func> | except:<func> | size:<n> | levels:<n>]

[lib:]<inlib>	Inline extracted functions from *inlib*
[name:]<func>	Inline function func
except:<func>	Do not inline function func
size:<n>	Inline only functions smaller than n statements (approximate)
levels:<n>	Inline n levels of functions

For C++ Codes, PGI Recommends IPA-based inlining or –Minline=levels:10!

Other C++ recommendations

- **Encapsulation, Data Hiding -** small functions, inline!
- **Exception Handling –** use –no_exceptions until 7.0
- **Overloaded operators, overloaded functions -** okay
- **Pointer Chasing -** -Msafeptr, restrict qualifer, 32 bits?
- **Templates, Generic Programming –** now okay
- **Inheritance, polymorphism, virtual functions –** runtime lookup or check, no inlining, potential performance penalties

SMP Parallelization

- **–Mconcur for auto-parallelization on multi-core**
  - **Compiler strives for parallel outer loops, vector SSE inner loops**
  - **–Mconcur=innermost forces a vector/parallel innermost loop**
  - **–Mconcur=cncall enables parallelization of loops with calls**
- **–mp to enable OpenMP 2.5 parallel programming model**
  - **See PGI User's Guide or OpenMP 2.5 standard**
  - **OpenMP programs compiled w/out –mp=nonuma**
- **–Mconcur and –mp can be used together!**

EKOPath Basic Optimizations

PathScale

`-g`	Generate debug (DWARF) information. Changes optimization level to -O0 unless explicitly overridden.
`-O0`	No optimization
`-O1`	Local optimization (straight line code)
`-O2`	Global scalar optimizations (-O2 is default)
`-O3`	Loop level transformations and vectorizations
`-ipa`	Inter-procedural optimizations (whole program). Can be used at any optimization level.
`-OPT:Ofast`	Generally safe but may impact floating point correctness. Maximizes performance. Equivalent to: `-OPT:ro=2:Olimit=0:div_split=ON:alias=typed`
`-Ofast`	Equivalent to `-O3 -ipa -OPT:Ofast -fno-math-errno`

Option Groups

PathScale

- Options organized into groups by compiler phase or by class of feature
- General syntax:

  `-GROUPNAME:opt[=val]{:opt=[val]}`
- Some GNU-style flags map to these options

  `-march -ffast-math -ffloat-store -fno-inline`
- Group names:

`-LIST:`	User listing
`-OPT:`	Optimizations
`-TARG:`	Target machine
`-TENV:`	Target environment
`-INLINE:`	Back-end inlining
`-IPA:`	Inter-procedural analysis
`-LANG:`	Language features
`-CG:`	Code generation
`-WOPT:`	Global scalar optimization
`-LNO:`	Loop nest optimization

Alias options

Improving performance of generated code by allowing the compiler to make assumptions about aliasing

Mainly for C/C++ programs

- -OPT:alias=typed
  - Activate ANSI/ISO C standard
  - Object not aliased if they have different base types
  - Implied by –Ofast
- -OPT:alias=restrict
  - Regard all pointers as having the 'restrict' attribute
- -OPT:alias=disjoint
  - No two pointers ever point to the same object
  - Many programs will not run correctly with this option

Controlling Floating-point Code

- Lower precision requirements to allow for faster code
  - `OPT:roundoff=`
    - Specifies extent of roundoff error the compiler is allowed to introduce
      - `0` = no roundoff error (default at `-O0`, `-O1`, `-O2`)
      - `1` = limited roundoff error (default at `-O3`)
      - `2` = allow roundoff error due to re-associating expressions (default at `-Ofast`)
      - `3` = any roundoff error is allowed
  - `OPT:IEEE_arithmetic=`
    - Specifies level of conformance to IEEE 754 floating-point roundoff and overflow behavior
      - 1 = string conformance to IEEE accuracy (default at `-O0`, `-O1`, `-O2`)
      - 2 = allow inexact results not conforming to IEEE 754 (default at `-O3`)
      - 3 = allow any mathematically valid transformations
  - `-OPT:IEEE_NaN_Inf=(on|off)`
    - Controls conformance to IEEE for Not-a-Number and Infinity operands
    - Default is `on`

Parallelization

PathScale™

- OpenMP
  - Option to enable directives:

    `-mp`
  - OpenMP 2.5 in Fortran, C, & C++
    - The C++ OpenMP support is limited and does not support OpenMP directives in C++ source that use exceptions, classes or templates. (We have found some codes with very simplistic use of classes may work.)

- Autoparallelization
  - Option to enable: `-apo`

Performance Tuning

-Ofast is the most aggressive optimization option

Equivalent to `-O3 -ipa -OPT:Ofast -fno-math-errno`

`-OPT:Ofast` is equivalent to

`-OPT:ro=2:Olimit=0:div_split=ON:alias=typed`

- A large number of other options are related to performance tuning
  - Phase specific options:

```
-CG
-INLINE
-IPA
-WOPT
-LNO
```

- PathOpt2 allows automatic search of best flag combinations

Tuning for AMD "Barcelona"

PathScale

- Tuned Prefetch for smaller L2 cache
  - option LNO: stream_prefetch=1
  - option CG:use_prefetch_nta (non temporal)
- SIMD unaligned loads
  - improves vectorization
- FP instruction tuning
  - aligned packed double (movapd)
    - replaces scalar double (movsd)
    - removes register dependency
  - movsd replaces movlpd (for loads)

Above changes account for >10% performance improvement

Options to Help Expose User Errors

- Programs may run incorrectly only at higher optimization levels
  - Causes include compiler bugs or bad coding practices

- To help diagnose bad coding practices
  - `-OPT:alias=no_parm`
    - Fortran compiler does NOT assume Fortran no-alias rule for parameters
  - `-LANG:rw_const=on`
    - For cases where callee modifies constant argument
  - `-trapuv`
    - Initializes variables with NaN. If program uses the uninitialized variable, it will crash instead of generating incorrect results
  - `-zerouv`
    - Initializes variables to 0
    - Good for programs that incorrectly assume memory is always initialized to zero.

The Cray Compiler Environment: Introduction and Intial Results

Nathan Wichmann
wichmann@cray.com

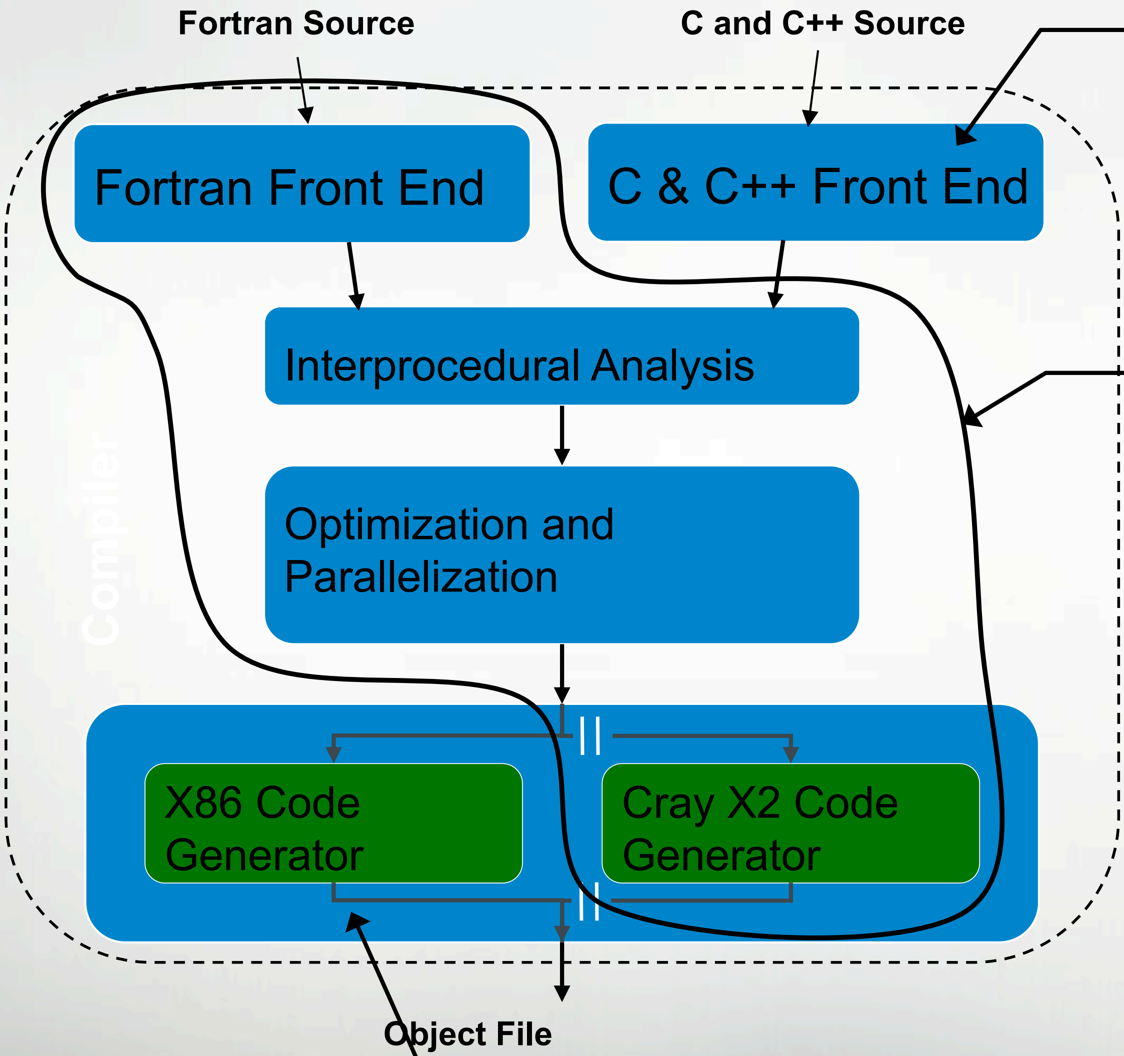
Outline

- Introduction to the Cray compiler
- Example
  - GTC
  - Overflow
  - PARQUET

Cray Compiler Environment (CCE): Brief History of Time

- Cray has a long tradition of high performance compilers
  - Vectorization
  - Parallelization
  - Code transformation
  - More…
- Began internal investigation leveraging an open source compiler called LLVM
- Initial results and progress better than expected
- Decided to move forward with Cray X86 compiler
- 7.0 released in December 2008
- 7.1 will be released Q2 2009

Technology Sources



C and C++ Front End supplied by Edison Design Group, with Cray-developed code for extensions and interface support

Cray Inc. Compiler Technology

X86 Code Generation from Open Source **LLVM**, with additional Cray-developed optimizations and interface support

Cray Opteron Compiler: How to use it

- Make sure it is available
  - module avail PrgEnv-cray
- To access the Cray compiler
  - module load PrgEnv-cray
- To target the Barcelona chip
  - module load xtpe-quadcore
- Once you have loaded the module “cc” and “ftn” are the Cray compilers
  - Recommend just using default options
  - Use –rm (fortran) and –hlist=m (C) to find out what happened
- man crayftn

Cray Opteron Compiler: Current Capabilities

- Excellent Vectorization
  - Vectorize more loops than other compilers
- OpenMP
  - 2.0 standard
  - Nesting
- PGAS: Functional UPC and CAF available today.
- Excellent Cache optimizations
  - Automatic Blocking
  - Automatic Management of what stays in cache
- Prefetching, Interchange, Fusion, and much more…

Cray Opteron Compiler: Future Capabilities

- C++ Support
- Automatic Parallelization
  - Modernized version of Cray X1 streaming capability
  - Interacts with OMP directives
- OpenMP 3.0
- Optimized PGAS
  - Will require Gemini network to really go fast
- Improved Vectorization
- Improve Cache optimizations

Case Study: The Gyrokinetic Toroidal Code (GTC)

- Plasma Fusion Simulation
- 3D Particle-in-cell code (PIC) in toroidal geometry
- Developed by Prof. Zhihong Lin (now at UC Irvine)
- Code has several different characteristics
  - Stride-1 copies
  - Strided memory operations
  - Computationally intensive
  - Gather/Scatter
  - Sorting and Packing
- Main routine is known as the "pusher"

Case Study: The Gyrokinetic Toroidal Code (GTC)

- Main Pusher kernel consists of 2 main loop nests
- First loop nest contains groups of 4 statements which include significant indirect addressing

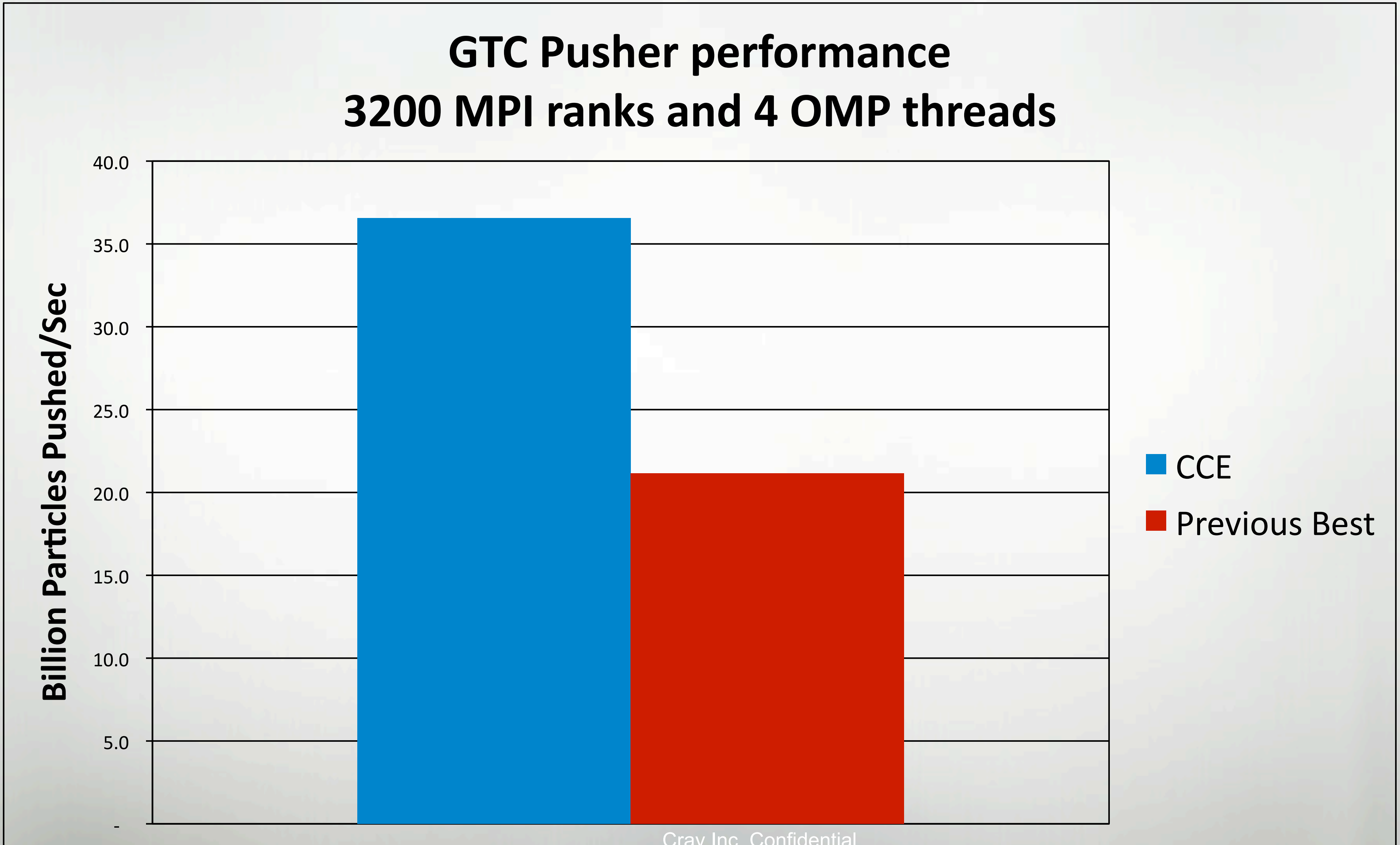
```
e1=e1+wp0*wt00*(wz0*gradphi(1,0,ij)+wz1*gradphi(1,1,ij))
e2=e2+wp0*wt00*(wz0*gradphi(2,0,ij)+wz1*gradphi(2,1,ij))
e3=e3+wp0*wt00*(wz0*gradphi(3,0,ij)+wz1*gradphi(3,1,ij))
e4=e4+wp0*wt00*(wz0*phit(0,ij)+wz1*phit(1,ij))
```

  - Turn 4 statements into 1 vector shortloop

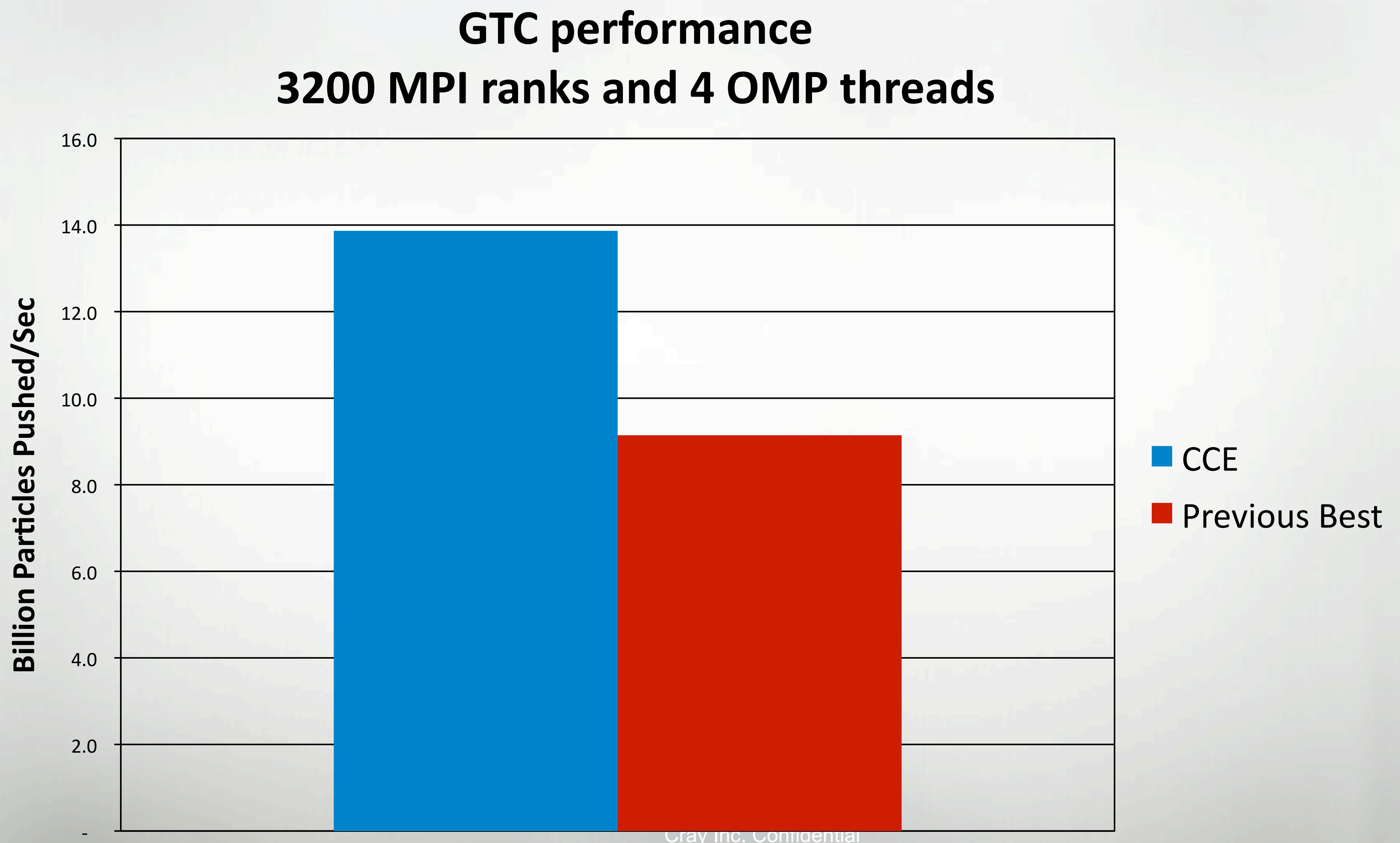
```
ev(1:4)=ev(1:4)+wp0*wt00*(wz0*tempphi(1:4,0,ij)+wz1*tempphi(1:4,1,ij))
```

- Second loop is large, computationally intensive, but contains strided loads and computed gather
  - CCE automatically vectorizes loop

Case Study: GTC

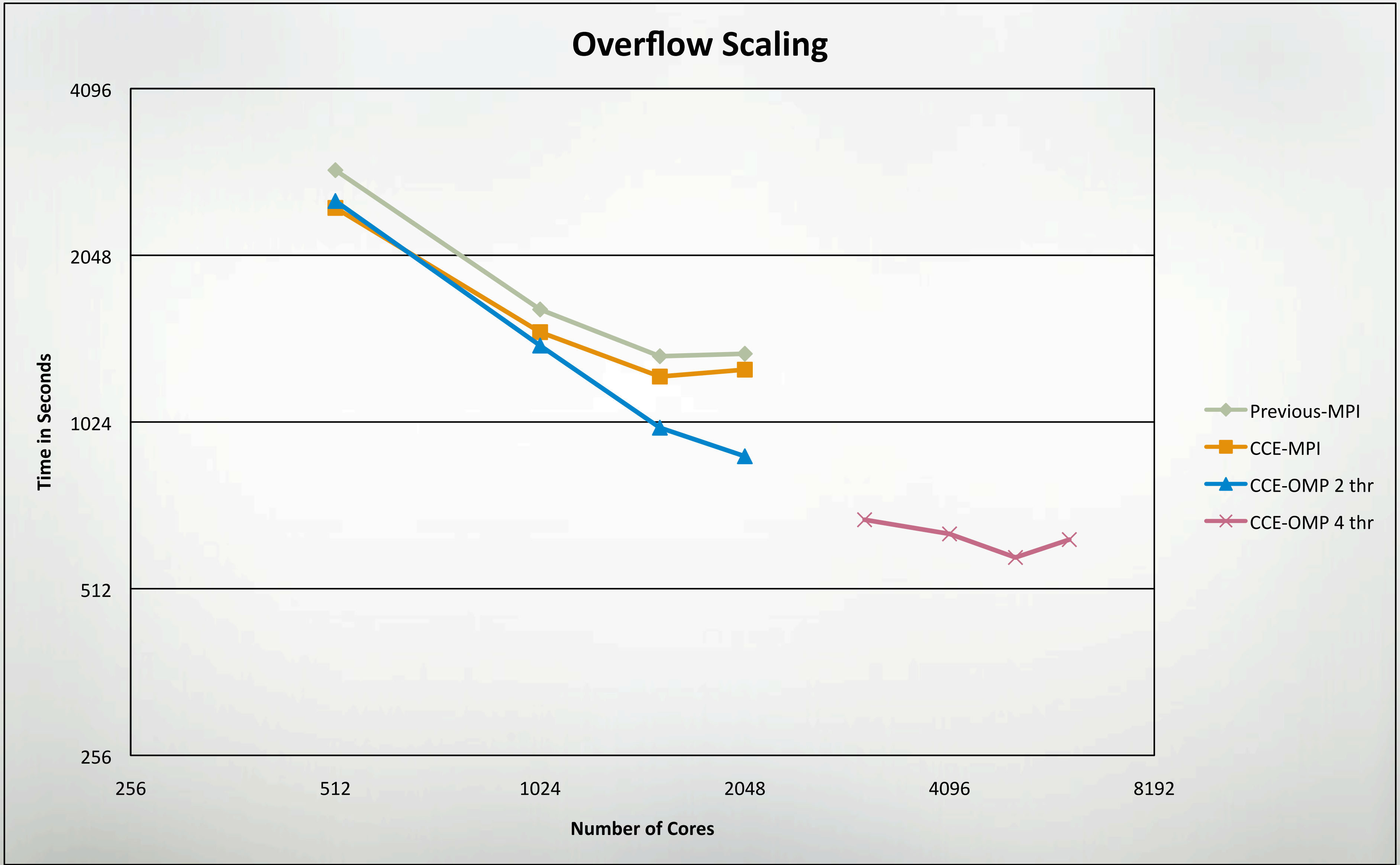


Case Study: GTC



- Overflow is a NASA developed Navier-Stokes flow solver for unstructured grids
- Subroutines consist of two or three simply-nested loops
- Inner loops tend to be highly vectorized and have 20-50 Fortran statements
- MPI is used for parallel processing
  - Solver automatically splits grid blocks for load balancing
  - Scaling is limited due to load balancing at > 1024
- Code is threaded at a high-level via OpenMP

Overflow Scaling using only MPI vs MPI & OMP



- Materials Science code
- Scales to 1000s of MPI ranks before it runs out of parallelism
- Want to use shared memory parallelism across entire node

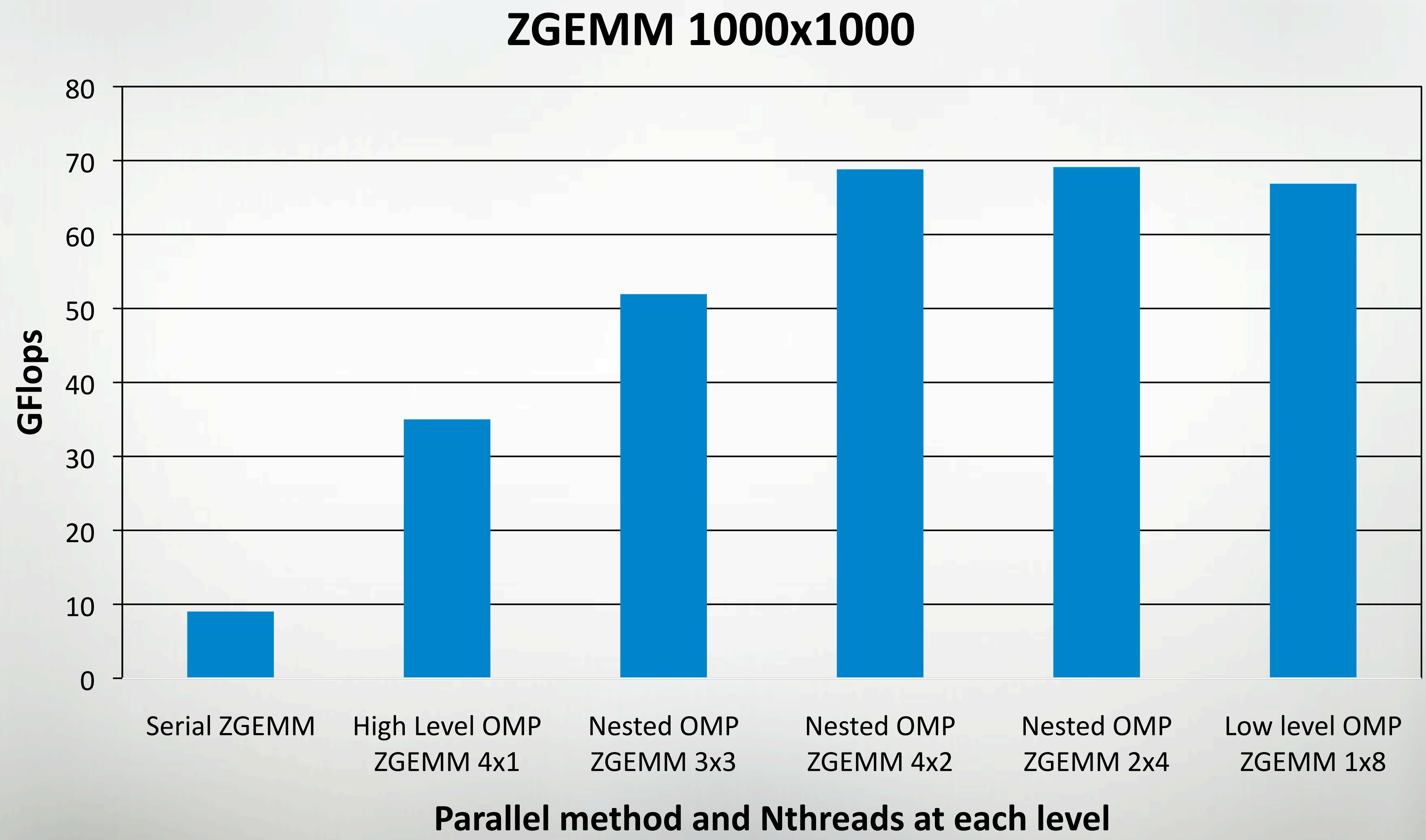
- Main kernel consists of 4 independent zgemms
- Want to use multi-level OMP to scale across the node

Case Study: PARQUET

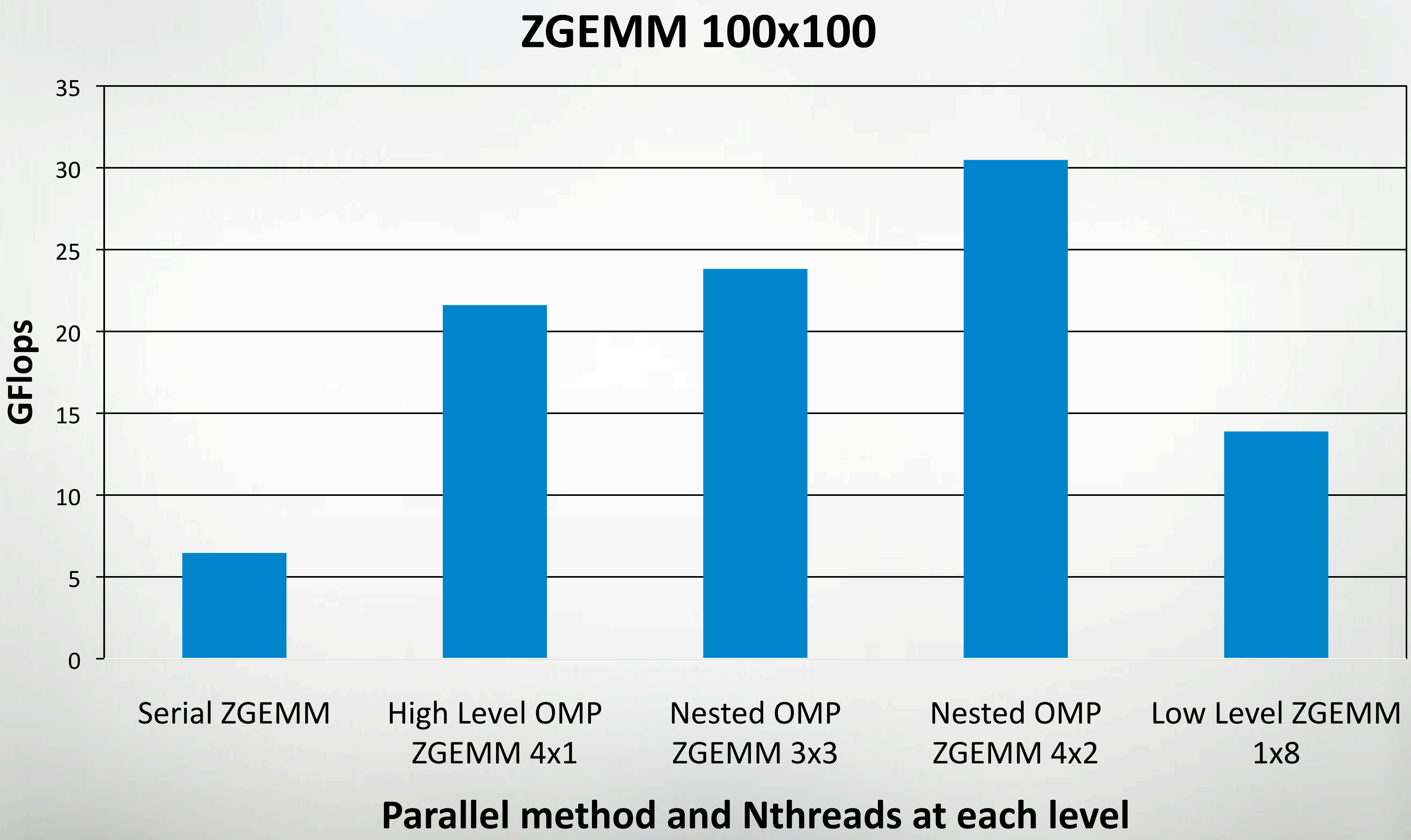
```
!$omp parallel do …
do i=1,4
   call complex_matmul(…)
enddo

Subroutine complex_matmul(…)
!$omp      parallel do private(j,jend,jsize)! num_threads(p2)
   do j=1,n,nb
     jend = min(n, j+nb-1)
     jsize = jend - j + 1
     call zgemm( transA,transB, m,jsize,k,                &
          alpha,A,ldA,B(j,1),ldb, beta,C(1,j),ldC)
   enddo
```

Case Study: PARQUET



Case Study: PARQUET



Conclusions

- The Cray Compiling Environment is a new, different, and interesting compiler with several unique capabilities
- Several codes are already taking advantage of CCE
- Development is ongoing
- Consider trying CCE if you think you could take advantage of its capabilities