Managing Code Transformations for Better Performance Portability

William D. Gropp
wgropp.cs.Illinois.edu

with
Thiago Teixeira and David Padua

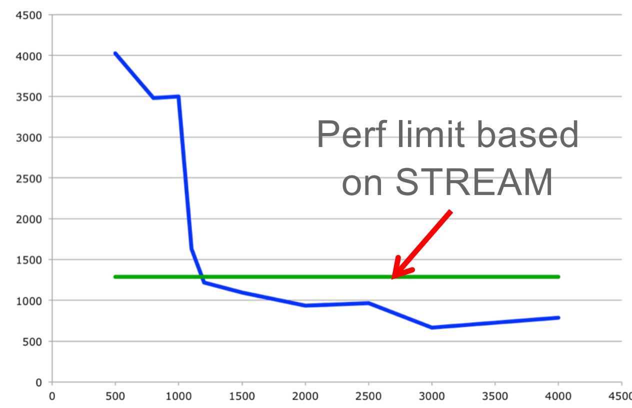
NCSA

Dreams and Reality

- For codes that demand performance (and parallelism almost always implies that performance is important enough to justify the cost and complexity of parallelism), the dream is performance portability
- The reality is that most codes require specialized code to achieve high performance, even for non-parallel codes
- A typical refrain is “Let The Compiler Do It”
  - This is the right answer …
    - If only the compiler *could* do it
  - Lets look at one of the simplest operations for a single core, dense matrix transpose
    - Transpose involves only data motion; no floating point order to respect
    - Only a double loop (fewer options to consider)

A Simple Example: Dense Matrix Transpose

-
```
do j=1,n
    do i=1,n
        b(i,j) = a(j,i)
    enddo
enddo
```
- No temporal locality (data used once)
- Spatial locality only if (words/cacheline) * n fits in cache

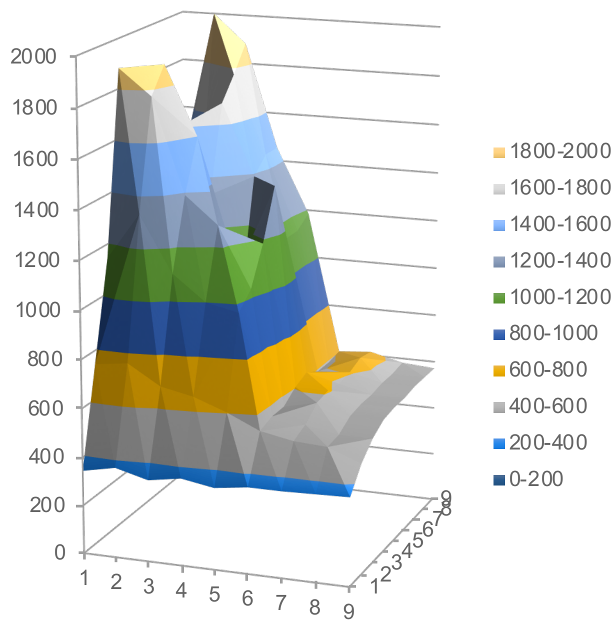


- Performance plummets when matrices no longer fit in cache

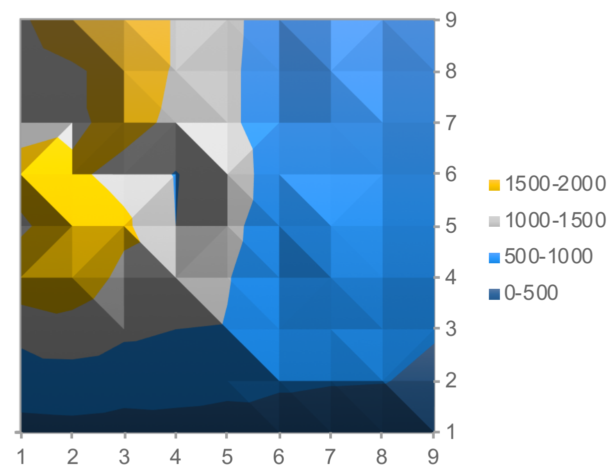
Blocking for cache helps

- ```
  do jj=1,n,stridej
     do ii=1,n,stridei
        do j=jj,min(n,jj+stridej-1)
           do i=ii,min(n,ii+stridei-1)
              b(i,j) = a(j,i)
  ```
- Good choices of stridei and stridej can improve performance by a significant factor
- How sensitive is the performance to the choices of stridei and stridej?
  ```

Results: Blue Waters O3



Simple, unblocked code compiled with O3 – 709MB/s



Real Codes Include Performance Workarounds

- Code excerpt from VecMDot_Seq in PETSc
- Code is unrolled to provide performance
  - Decision was made once (and verified as worth the effort *at the time*)
  - Remains part of the code forevermore
  - Unroll by 4 *probably* good for vectorization
    - But not necessarily best for performance
    - Does not address alignment

```
switch (j_rem=j&0x3) {
case 3:
  x2     = x[2];
  sum0 += x2*yy0[2]; sum1 += x2*yy1[2];
  sum2 += x2*yy2[2];
case 2:
  x1     = x[1];
  sum0 += x1*yy0[1]; sum1 += x1*yy1[1];
  sum2 += x1*yy2[1];
case 1:
  x0     = x[0];
  sum0 += x0*yy0[0]; sum1 += x0*yy1[0];
  sum2 += x0*yy2[0];
case 0:
  x    += j_rem;
  yy0 += j_rem;
  yy1 += j_rem;
  yy2 += j_rem;
  j    -= j_rem;
  break;
}
while (j>0) {
  x0 = x[0];
  x1 = x[1];
  x2 = x[2];
  x3 = x[3];
  x += 4;

  sum0 += x0*yy0[0] + x1*yy0[1] + x2*yy0[2] + x3*yy0[3]; yy0+=4;
  sum1 += x0*yy1[0] + x1*yy1[1] + x2*yy1[2] + x3*yy1[3]; yy1+=4;
  sum2 += x0*yy2[0] + x1*yy2[1] + x2*yy2[2] + x3*yy2[3]; yy2+=4;
  j    -= 4;
}
z[0] = sum0;
z[1] = sum1;
z[2] = sum2;
```

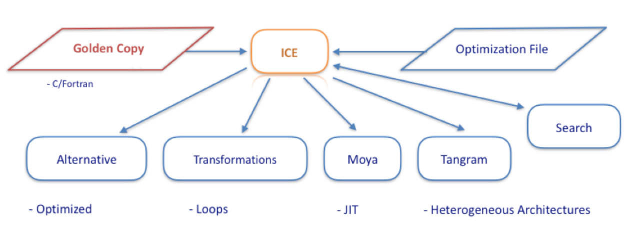
Practical Performance Optimization

- How to handle all the required optimizations together for many different scenarios?
- How to keep the code maintainable?
- How to find the best sequence of optimizations?

- Requirements
  - ”Golden Copy” code runs without ICE – do not require “buy in” to the system
  - Permit incremental adoption – apply ICE to subsets of the code, with subsets of tools
  - Coexist with other tools
  - Separate generation of optimized code from develop/run so that users do not need to install/run those tools. Allow tuning runs on “related” systems (e.g., x86 vectorization)
  - Support ways to find the best sequence of optimizations

Illinois Coding Environment (ICE)

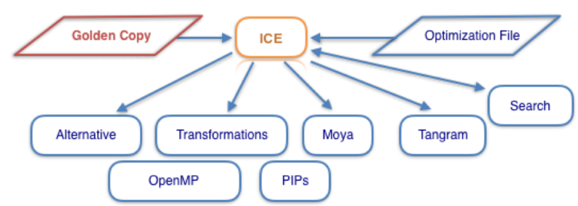
- One pragmatic approach
- Assumptions
  - Fast code requires some expert intervention
  - Can't all be done at compile time
  - Original code (in standard language) is maintained as reference
  - Can add information about computation to code
- Center for Exascale Simulation of Plasma-Coupled Combustion
  - http://xpacc.illinois.edu
  - ICE used to support "Golden Copy" code – version natural for computational scientist, without code optimizations
  - Used with primary simulation code, PlasCom2



- Approach
  - Annotations provide additional descriptive information
    - Block name, expected loop sizes, etc.
  - Source-to-source transformations used to create code for compiler
    - Exploit tool ecosystem – interface to existing tools
    - Original "Golden Copy" used for development, correctness checks
  - Database used to manage platform-specific versions; detect changes that invalidate transformed versions
    - Don't need to install/run transformation tools

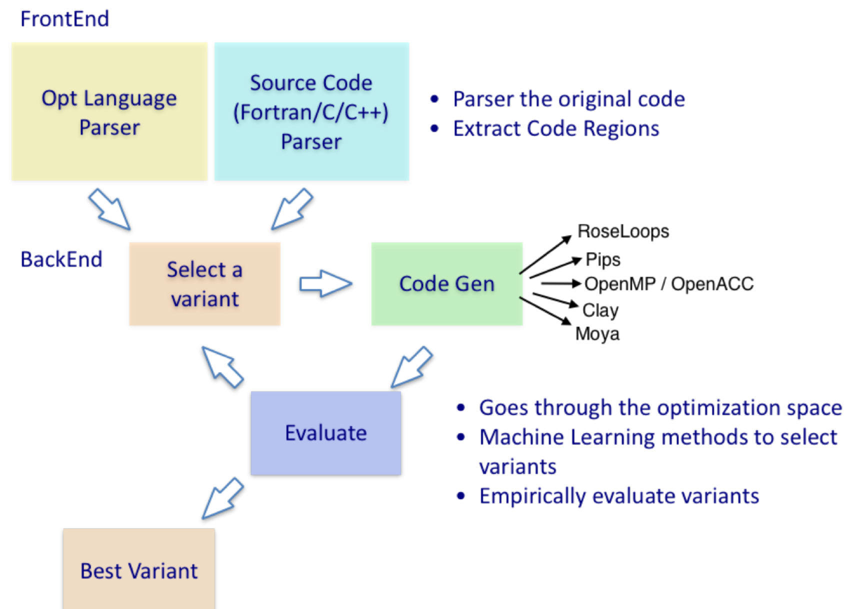
ICE

- Golden copy approach: baseline version without architecture- or compiler-specific optimizations (not buy-in)
- Search combined with application's developer expertise
- Build-time, Compile-time and Runtime optimizations
- Non-prescriptive, Gradual adoption, Separation of Concerns
- Reuse of other optimizations tools already implemented
  - Interfaces to simplify plug-in
- Search and optimization tools



ICE

- Source code is annotated to define code regions
- Optimization file notation orchestrates the use of the optimization tools on the code regions defined
- Interface provides operations on the Source code to invoke optimizations through:
  - Adding pragmas
  - Adding labels
  - Replacing code regions
- These operations are used by the interface to plug-in optimization tools
- Most tools are source-to-source
  - tools must understand output of previous tools



Matrix Multiplication Example

```
#pragma @ICE loop=matmul
        for (i=0; i<matSize; i++)
            for (j=0; j<matSize; j++) {
               for (k=0; k<matSize; k++) {
                    matC[i][j] += matA[i][k] * matB[k][j];
               }
            }
        }
```

```
—
# Built command before compilation
prebuildcmd:

# Compilation command before tests
buildcmd: make realclean;  make

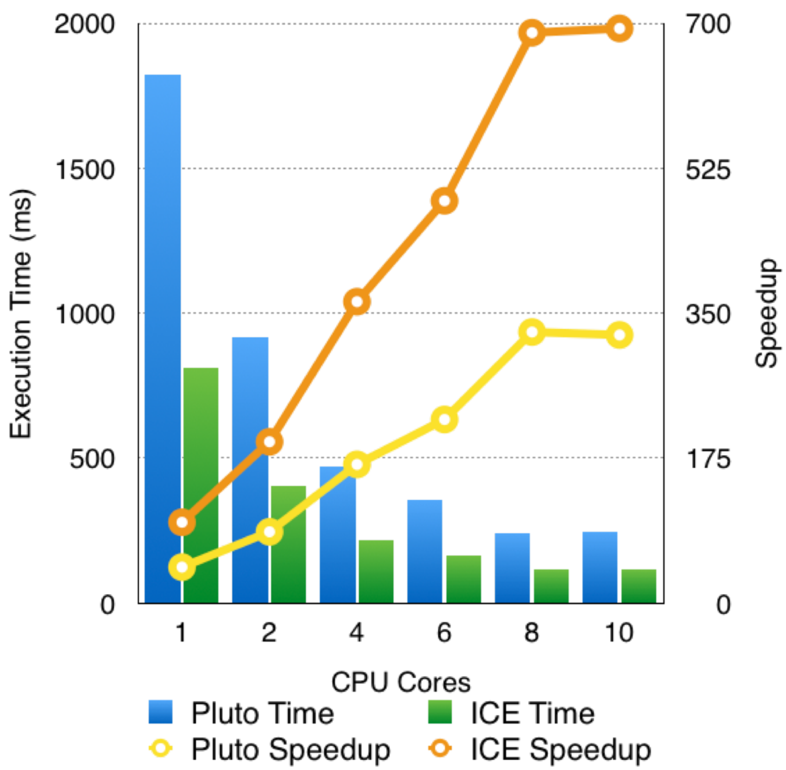
#Command call for each test
runcmd: ./mmc

matmul:
    - Pips.tiling+:
 loop: 1
        factor: [2..512, 2..512, 2..512]
    - Pips.tiling+:
        loop: 4
        factor: [8, 16, 8]
    - OpenMP.OMPFor+:
loop: 1

…
```

```
 #pragma omp parallel for schedule(static,1) private(i_t, k_t, j_t,i_t_t, k_t_t
,j_t_t, i, k,j)
  for (i_t = 0; i_t <= 127; i_t += 1)
    for (k_t = 0; k_t <= 127; k_t += 1)
    for (j_t = 0; j_t <= 3; j_t += 1)
    for (i_t_t = 4 * i_t; i_t_t <= ((4 * i_t) + 3); i_t_t += 1)
    for (k_t_t = 2 * k_t; k_t_t <= ((2 * k_t) + 1); k_t_t += 1)
    for (j_t_t = 32 * j_t; j_t_t <= ((32 * j_t) + 31); j_t_t += 1)
    for (i = 4 * i_t_t; i <= ((4 * i_t_t) + 3); i += 1)
    for (k = 8 * k_t_t; k <= ((8 * k_t_t) + 7); k += 1)
    for (j = 16 * j_t_t; j <= ((16 * j_t_t) + 15); j += 1)
    matC[i][j] += matA[i][k] * matB[k][j];
```

Matrix Multiplication Results



- Two levels of tiling + OpenMP
- Original version: 78,825 ms
- 98x speedup (1 core)
- 694x speedup (10 cores)
- Avg 2.2x speedup over Pluto

2048^2 ELEMENTS
ICC 17.0.1
INTEL E5-2660 V3
PLUTO PET BRANCH

NCSA

Stencil 3D

```
#pragma @ICE loop=stencil
for(i = 1; i < x-1; i++) {
  for(j = 1; j < y-1; j++) {
    for(k = 1; k < z-1; k++) {
      B[i][j][k] = C0 * A[i][j][k] + C1 * (
                   A[i+1][j][k] + A[i-1][j][k] +
                   A[i][j+1][k] + A[i][j-1][k] +
                   A[i][j][k+1] + A[i][j][k-1]);
    }
  }
}
#pragma @ICE endloop
```

+

```
---
#Built command before compilation
prebuildcmd:

#Compilation command before tests
buildcmd:
    make realclean;  make CC={compiler} COPT={params}

buildoptions:
   gcc:
        params:{'-O':{'default': 3,'min': 0,'max': 3}}
   icc:
        params:{'-O':{'default': 3,'min': 0,'max': 3}}

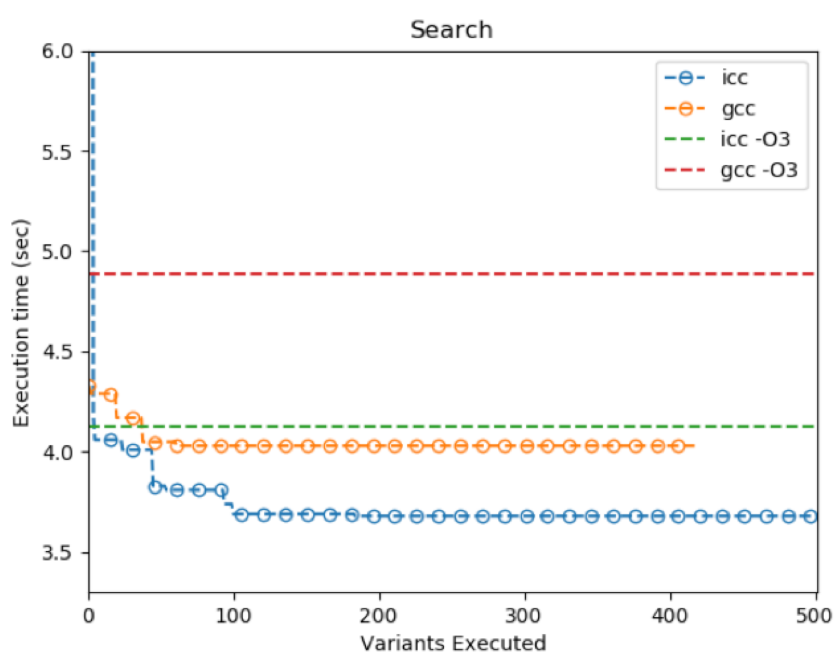
#Command call for each test
runcmd: ./sten3d 1024 20

tuning: on

stencil:
       rose_uiuc:
          - stripmine+:
              loop: 4
              factor: 16..1024
                type: poweroftwo
          - stripmine+:
              loop: 3
              factor: 16..1024
                type: poweroftwo
          - stripmine+:
              loop: 2
              factor: 16..1024
                type: poweroftwo
          - interchange+:
              order:0,1,3,5,2,4,6
...
```

Performance Results

- 3-D Stencil
  - 11,664 variants
  - Max 12.6 sec
  - Min 3.68 sec
  - Speedup over simple code
    - icc: 1.12x
    - gcc: 1.21x



Why No Example of Transpose?

- A lesson in why it is critical to separate code generation from everyday use of the optimized code
- Installing ICE and its full toolset is challenging
  - ICE uses pip to install required external packages
    - Good that ICE uses existing tools
    - Bad that MacOS version of pip is so old that it can't update itself
      - When did software engineering stop considering backward compatibility for more than a few months?
  - ICE uses rose to parse code
    - After downloading rose and associated tools (and the Java JDK, which was not where the Oracle web pages said it was), rose failed to build.
    - "I don't think you should try Rose on Mac. I've tried that before and couldn't pull it off."
    - Medium term fix – rose moving to use clang (?)
    - Short term fix – run ICE on Linux
- Real lesson – these tools are complex and fragile. ICE helps by providing a way to separate the process of creating the code transformations and using those transformations, while retaining "friendly" code

Conclusions

- It is often necessary to apply specific, system- and problem-dependent optimizations to the source code to achieve high performance
- ICE:
  - Separation of Concerns (opt file) +
  - Coexistence with other tools +
  - Gradual adoption +
  - Empirical search + Developer Knowledge
- Golden copy: the developer can focus on the problem
- Simple and easy to be used by the programmers
- Hard to get the tools to work though!

This material is based in part upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374.