

A Source-To-Source Approach to HPC Challenges

Dr. Chunhua “Leo” Liao

March 13th, 2015



LLNL-PRES- 668361

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

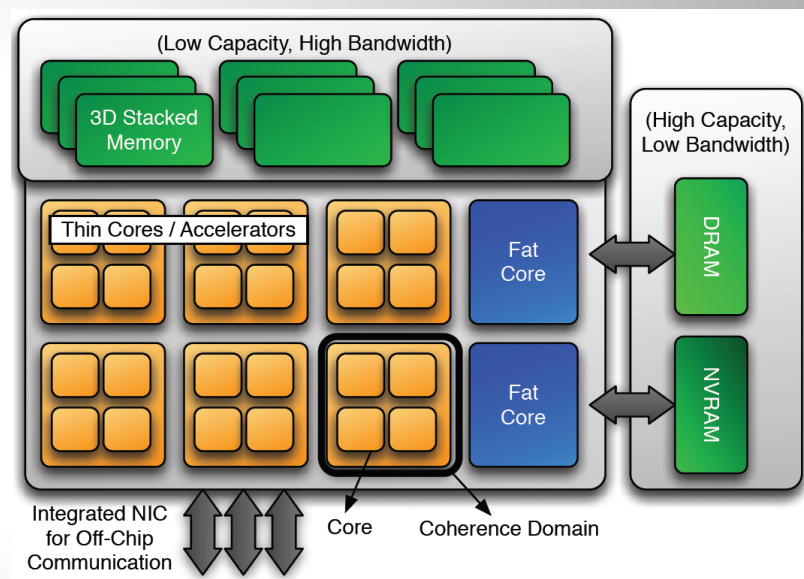


Agenda

- Background
 - HPC challenges
 - Source-to-source compilers
- Source-to-source compilers to address HPC challenges
 - Programming models for heterogeneous computing
 - Performance via autotuning
 - Resilience via source level redundancy
- Conclusion
- Future Interests

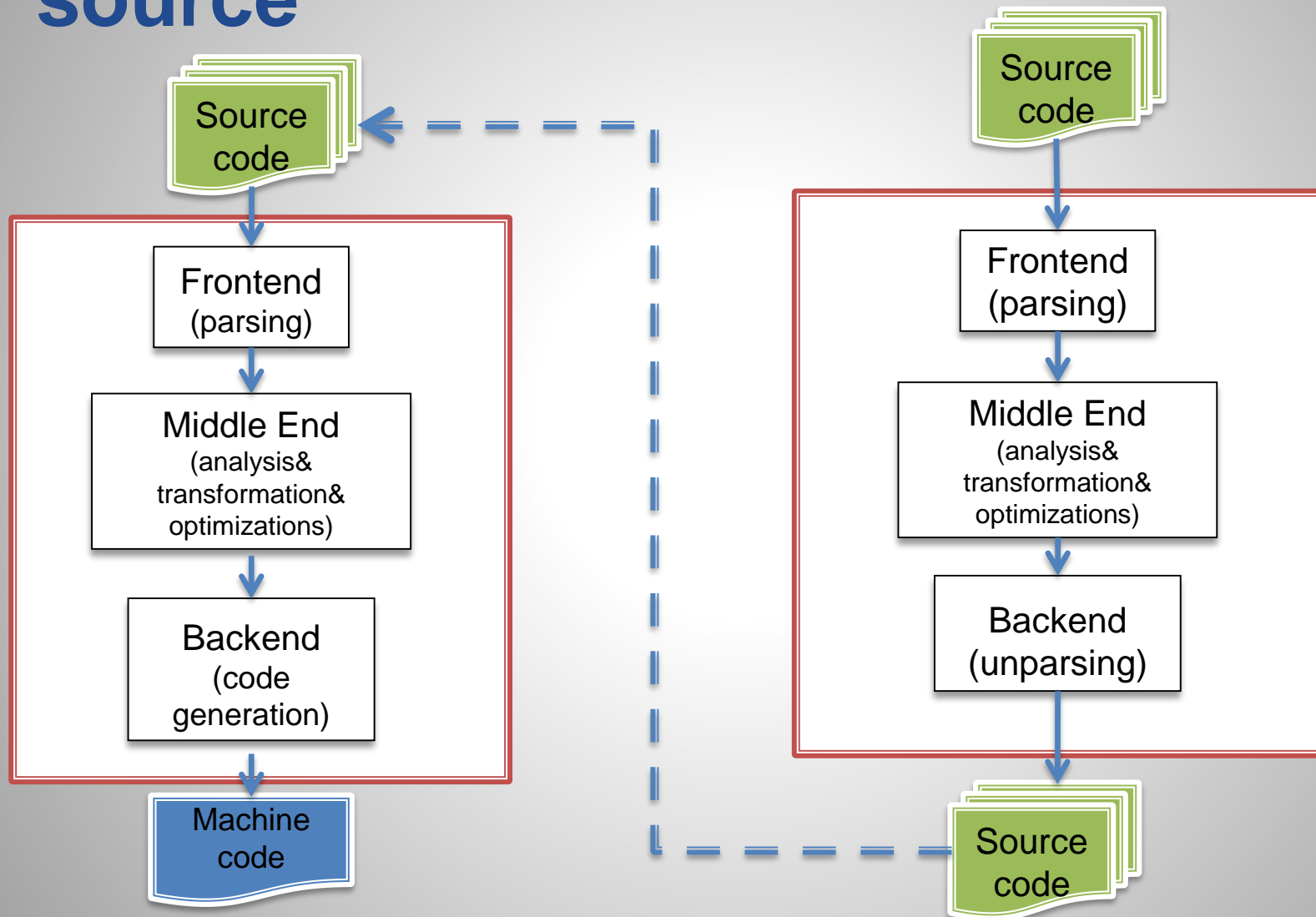
HPC Challenges

- HPC: High Performance Computing
 - DOE applications: climate modeling, combustion, fusion energy, biology, material science, etc.
 - Network connected (clustered) computation nodes
 - Increasing node complexity
- Challenges
 - Parallelism, performance, productivity, portability
 - Heterogeneity, power efficiency, resilience



J. A. Ang, et al. Abstract machine models and proxy architectures for exascale (10^{19} FLOPS) computing. Co-HPC '14

Compilers: traditional vs. source-to-source



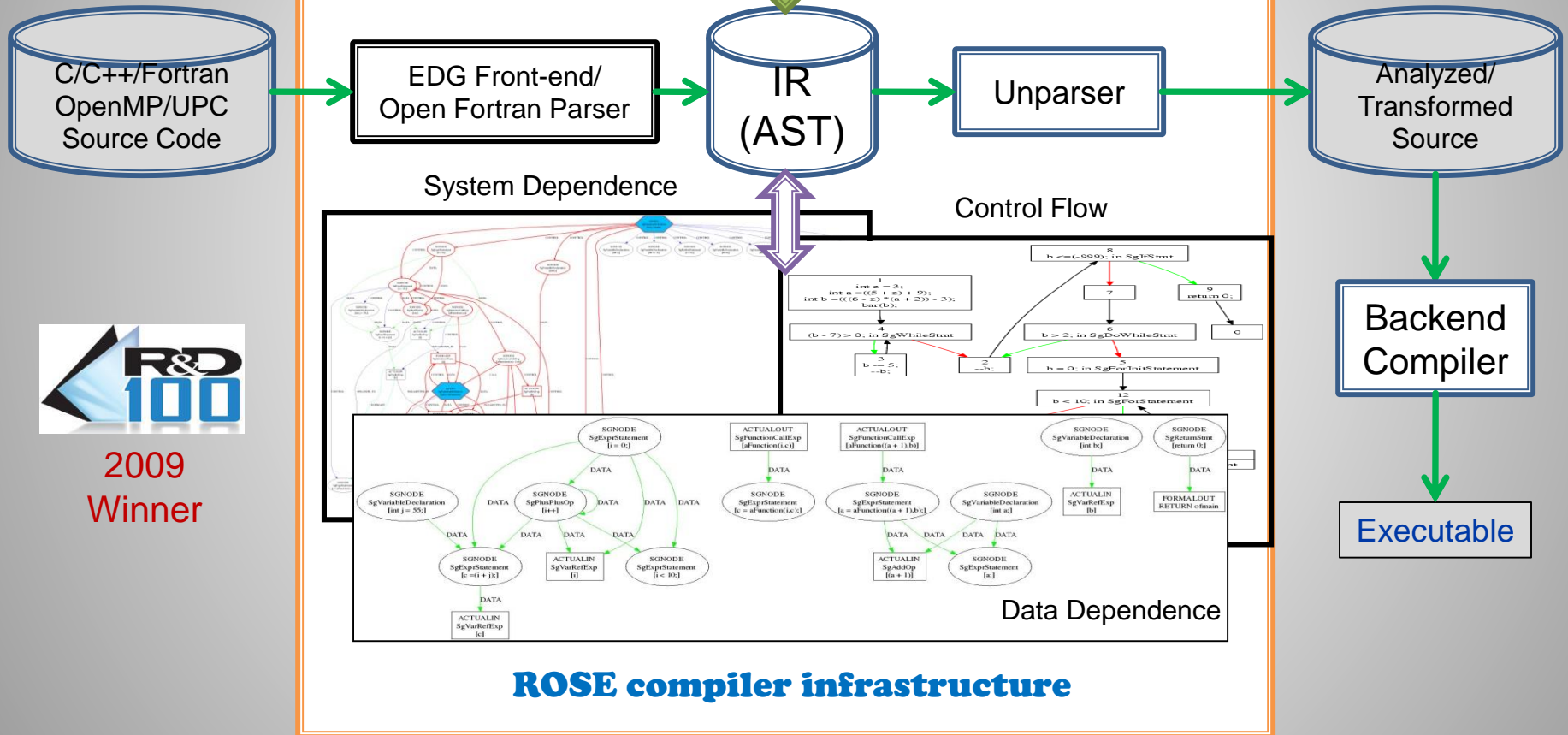
Why Source-to-Source?

	Traditional Compilers (GCC, ICC, LLVM..)	Source-to-source compilers (ROSE, Cetus, ..)
Goal	Use compiler technology to generate fast & efficient executables for a target platform	Make compiler technology accessible, by enabling users to build customized source code analysis and transformation compilers/tools
Users	Programmers	Researchers and developers in compiler, tools, and application communities.
Input	Programs in multiple languages	Programs in multiple languages
Internal representation	Multiple levels of Intermediate representations, losing source code structure and information	Abstract Syntax Tree, keeping all source level structure and information (comments, preprocessing info. etc), extensible.
Analyses & optimizations	Internal use only, black box to users	Individually exposed to users , composable and customizable
Output	Machine executables, hard to read	Human-readable , compilable source code
Portability	Multiple code generators for multiple platforms	Single code generator for all platforms, leveraging traditional backend compilers

ROSE Compiler Infrastructure

ROSE-based source-to-source tools

www.roseCompiler.org



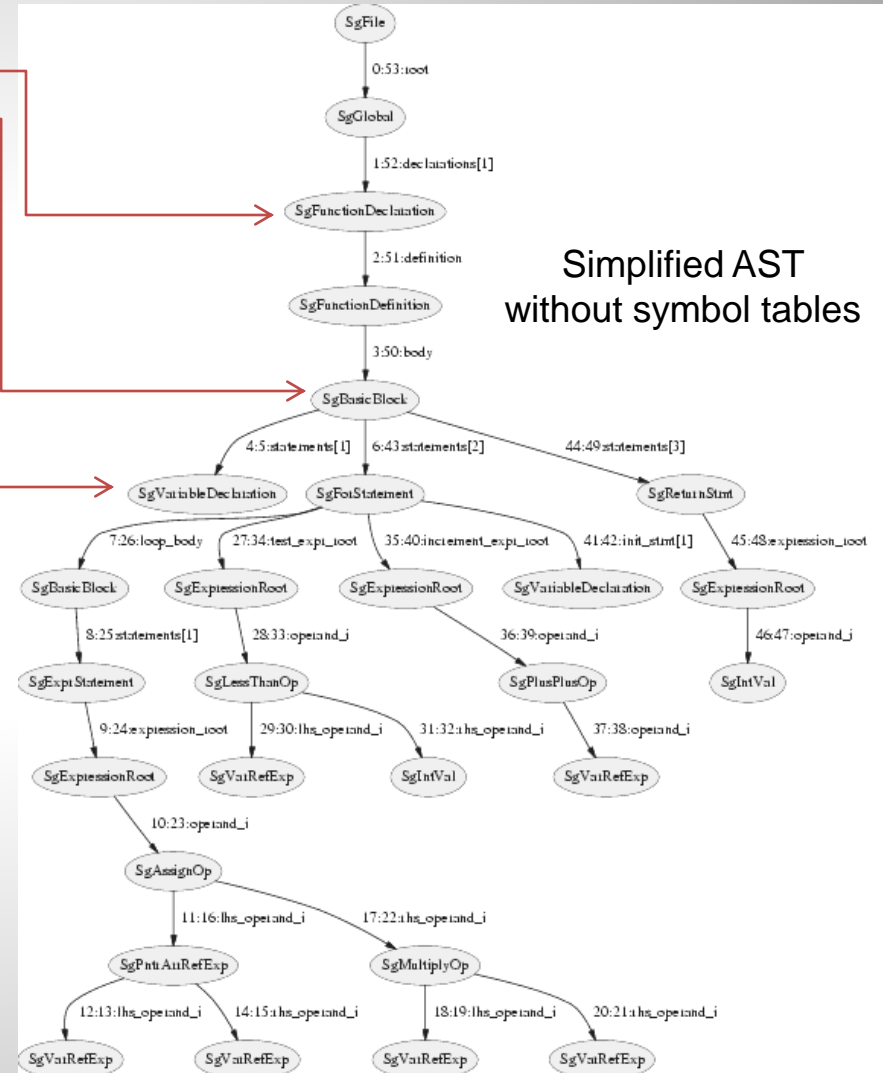
2009
Winner

ROSE IR = AST + symbol tables + interface

```
int main()
{
    int a[10];

    for(int i=0;i<10;i++)
        a[i]=i*i;

    return 0;
}
```



ROSE AST(abstract syntax tree)

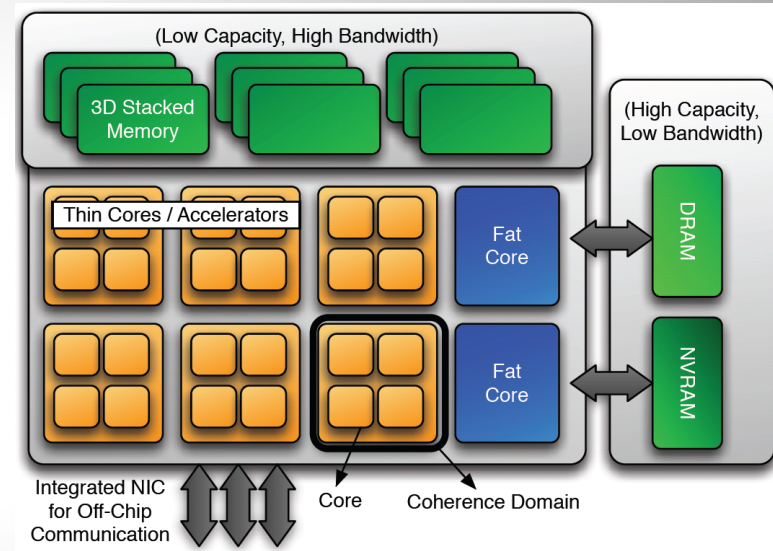
- High level: preserve all source level details
- Easy to use: rich set of interface functions for analysis and transformation
- Extensible: user defined persistent attributes

Using ROSE to help address HPC challenges

- Parallel programming model research
 - MPI, OpenMP, OpenACC, UPC, Co-Array Fortran, etc.
 - Domain-specific languages, etc.
- Program optimization/translation tools
 - Automatic parallelization to migrate legacy codes
 - Outlining and parameterized code generation to support auto tuning
 - Resilience via code redundancy
 - Application skeleton extraction for Co-Design
 - Reverse computation for discrete event simulation
 - Bug seeding for testing
- Program analysis tools
 - Programming styles and coding standard enforcement
 - Error checking for serial, MPI and OpenMP programs
 - Worst execution time analysis

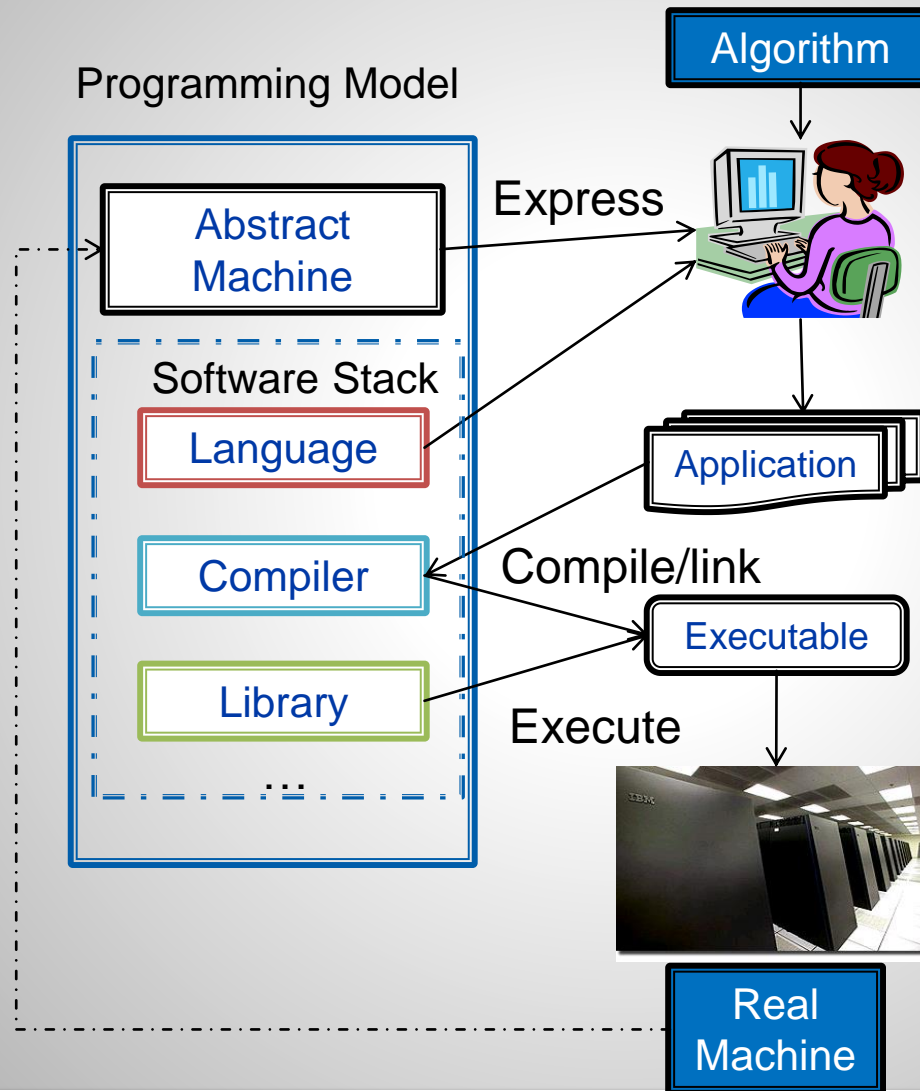
Case 1: high-level programming models for heterogeneous computing

- Problem
 - Heterogeneity challenge of exascale
 - Naïve approach: CPU code+ accelerator code + data transferring in between
- Approach
 - Develop high-level, directive based programming models
 - Automatically generate low level accelerator code
 - Prototype and extend the OpenMP 4.0 accelerator model
 - Result: HOMP (Heterogeneous OpenMP) compiler



J. A. Ang, et al. Abstract machine models and proxy architectures for exascale (10^{19} FLOPS) computing. Co-HPC '14

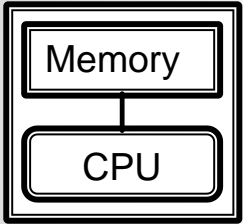
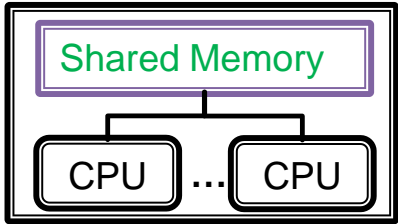
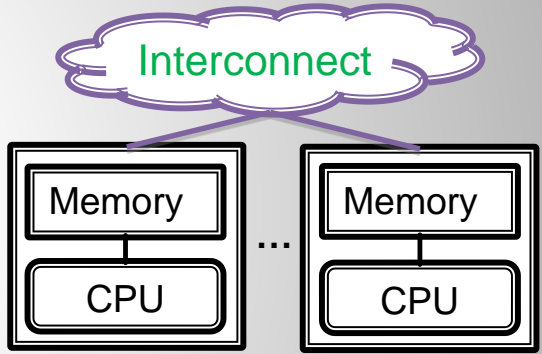
Programming models bridge algorithms and machines and are implemented through components of software stack



Measures of success:

- Expressiveness
- Performance
- Programmability
- Portability
- Efficiency
- ...

Parallel programming models are built on top of sequential ones and use a combination of language/compiler/library support

Programming Model	Sequential	Parallel	
		Shared Memory (e.g. OpenMP)	Distributed Memory (e.g. MPI)
Abstract Machine (overly simplified)			
Software Stack	General purpose Languages (GPL) C/C++/Fortran	GPL + Directives	GPL + Call to MPI libs
	Sequential Compiler	Seq. Compiler + OpenMP support	Seq. Compiler
	Optional Seq. Libs	OpenMP Runtime Lib	MPI library

OpenMP Code

```
#include <stdio.h>
```

```
#ifdef _OPENMP
```

```
#include <omp.h>
```

```
#endif
```

```
int main(void)
```

```
{
```

```
#pragma omp parallel
```

```
printf("Hello,world! \n");
```

```
return 0;
```

```
}
```

```
...
```

```
//PI calculation
```

```
#pragma omp parallel for reduction (+:sum) private (x)
```

```
for (i=1;i<=num_steps;i++)
```

```
{
```

```
    x=(i-0.5)*step;
```

```
    sum=sum+ 4.0/(1.0+x*x);
```

```
}
```

```
pi=step*sum;
```

```
..
```

Programmer

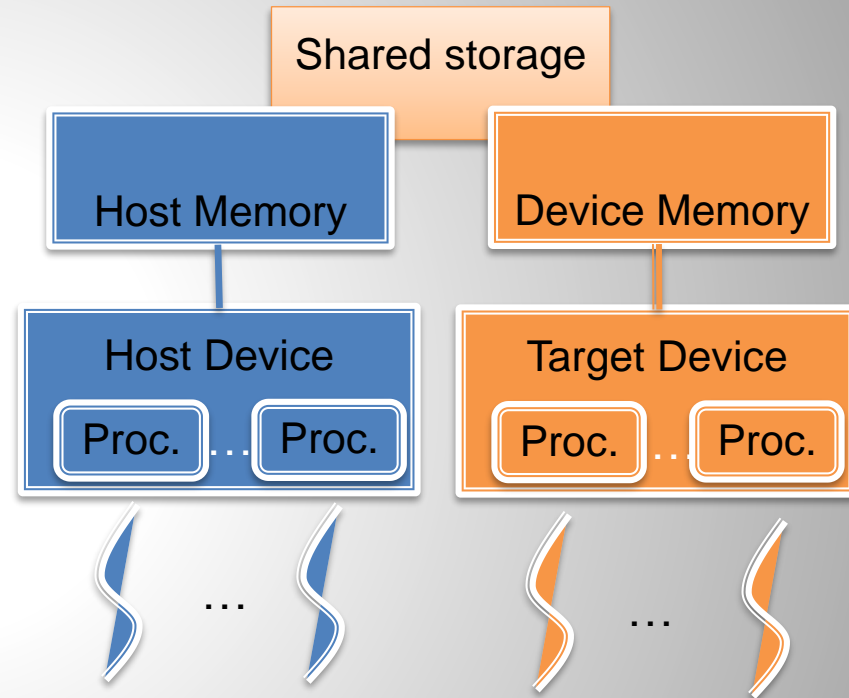
→ compiler

→ Runtime library

→ OS threads

OpenMP 4.0's accelerator model

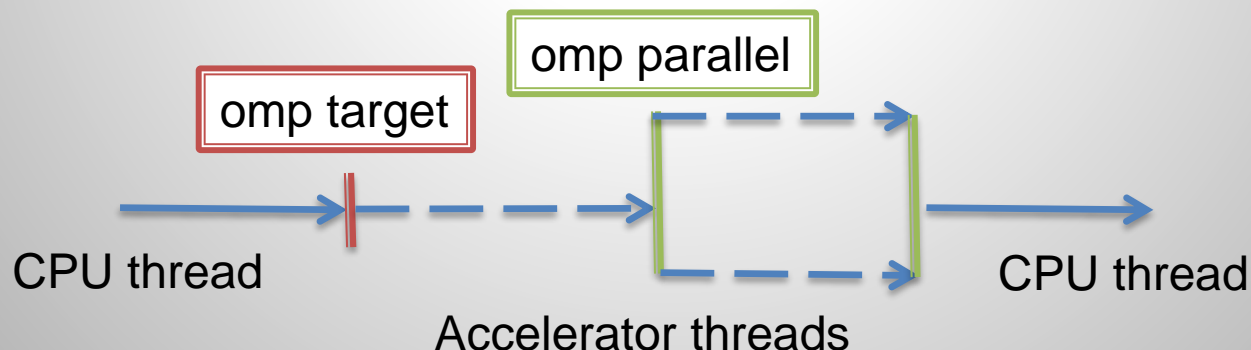
- Device: a logical execution engine
 - Host device: where OpenMP program begins, one only
 - Target devices: **1 or more** accelerators
- Memory model
 - Host data environment: one
 - Device data environment: one or more
 - Allow shared host and device memory
- Execution model: Host-centric
 - Host device : “offloads” code regions and data to target devices
 - Target devices: fork-join model
 - Host waits until devices finish
 - Host executes device regions if no accelerators are available /supported



OpenMP 4.0 Abstract Machine

Computation and data offloading

- Directive: `#pragma omp target device(id) map() if()`
 - **target**: create a device data environment and offload computation to be run in sequential on the same device
 - **device (int_exp)**: specify a target device
 - **map(to|from|tofrom|alloc:var_list)** : data mapping between the current task's data environment and a device data environment
- Directive: `#pragma omp target data device(id) map() if()`
 - Create a device data environment



Example code: Jacobi

```
#pragma omp target data device (gpu0) map(to:n, m, omega, ax, ay, b, \
    f[0:n][0:m]) map(tofrom:u[0:n][0:m]) map(alloc:uold[0:n][0:m])

while ((k<=mits)&&(error>tol))
{
    // a loop copying u[][] to uold[][] is omitted here
    ...
    #pragma omp target device(gpu0) map(to:n, m, omega, ax, ay, b, f[0:n][0:m], \
        uold[0:n][0:m]) map(tofrom:u[0:n][0:m])
    #pragma omp parallel for private(resid,j,i) reduction(+:error)
    for (i=1;i<(n-1);i++)
        for (j=1;j<(m-1);j++)
        {
            resid = (ax*(uold[i-1][j] + uold[i+1][j])\
                + ay*(uold[i][j-1] + uold[i][j+1]) + b * uold[i][j] - f[i][j])/b;
            u[i][j] = uold[i][j] - omega * resid;
            error = error + resid*resid ;
        } // the rest code omitted ...
    }
```

Building the accelerator support

- Language: OpenMP 4.0 directives and clauses
 - target, device, map, collapse, target data, ...
- Compiler:
 - CUDA kernel generation: outliner
 - CPU code generation: kernel launching
 - Loop mapping (1-D vs. 2-D), loop transformation (normalize, collapse)
 - Array linearization
- Runtime:
 - Device probing, configure execution settings
 - Loop scheduler: static even, round-robin
 - Memory management : data, allocation, copy, deallocation, reuse
 - Reduction: contiguous reduction pattern (folding)

Example code: Jacobi using OpenMP accelerator directives

```
#pragma omp target data device (gpu0) map(to:n, m, omega, ax, ay, b, \
    f[0:n][0:m]) map(tofrom:u[0:n][0:m]) map(alloc:uold[0:n][0:m])

while ((k<=mits)&&(error>tol))
{
    // a loop copying u[][] to uold[][] is omitted here
    ...
    #pragma omp target device(gpu0) map(to:n, m, omega, ax, ay, b, f[0:n][0:m], \
        uold[0:n][0:m]) map(tofrom:u[0:n][0:m])
    #pragma omp parallel for private(resid,j,i) reduction(+:error)
    for (i=1;i<(n-1);i++)
        for (j=1;j<(m-1);j++)
        {
            resid = (ax*(uold[i-1][j] + uold[i+1][j])\
                + ay*(uold[i][j-1] + uold[i][j+1]) + b * uold[i][j] - f[i][j])/b;
            u[i][j] = uold[i][j] - omega * resid;
            error = error + resid*resid ;
        } // the rest code omitted ...
    }
```

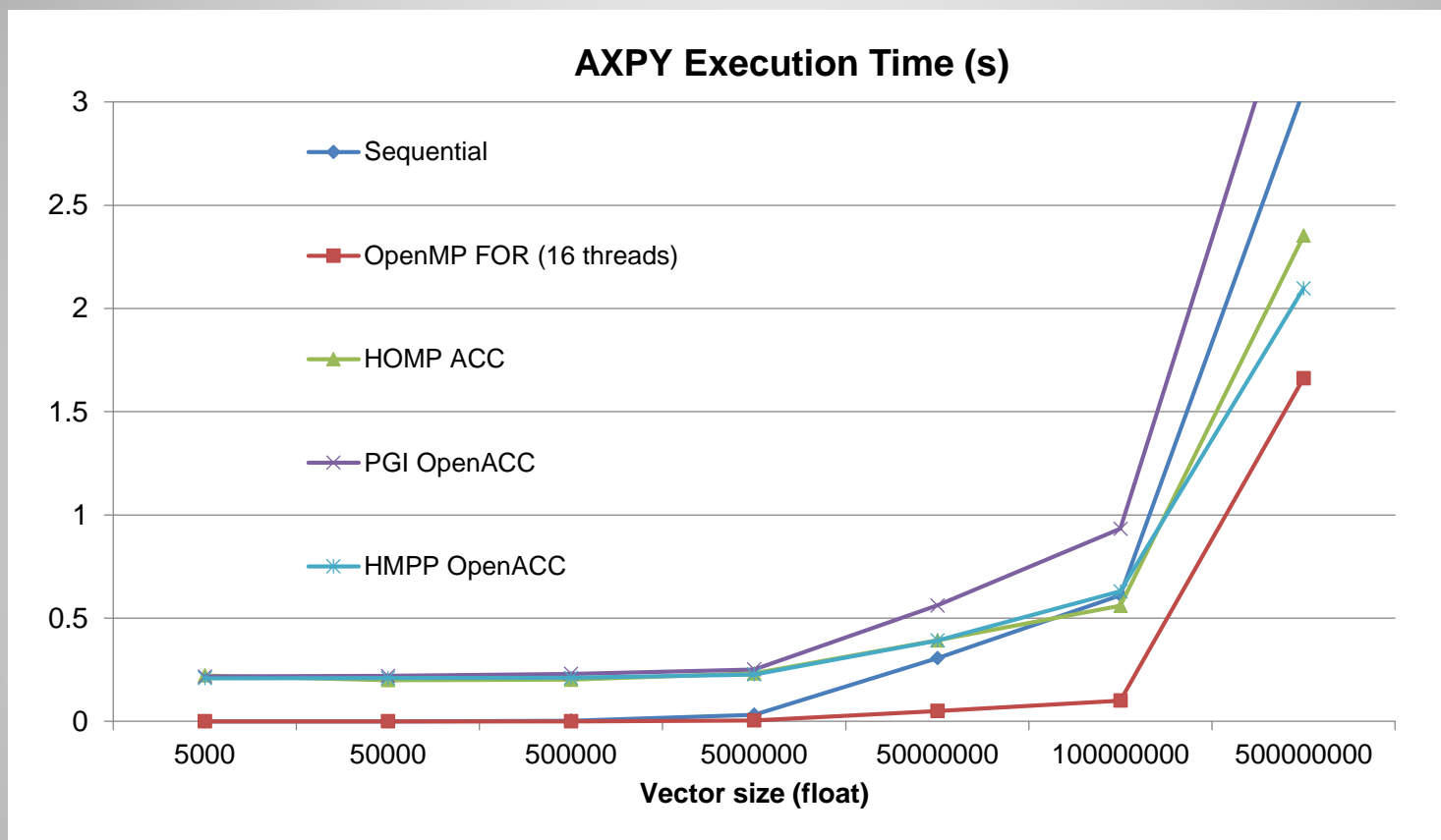
GPU kernel generation: outliner, scheduler, array linearization, reduction

```
1.  __global__ void OUT__1__10117__(int n, int m, float omega, float ax, float ay, float b, \
2.      float *_dev_per_block_error, float *_dev_u, float *_dev_f, float *_dev_uold)
3.  { /* local variables for loop , reduction, etc */
4.      int _p_j;  float _p_error;  _p_error = 0;  float _p_resid;
5.      int _dev_i, _dev_lower, _dev_upper;
6.      /* Static even scheduling: obtain loop bounds for current thread of current block */
7.      XOMP_accelerator_loop_default(1, n-2, 1, &_amp_dev_lower, &_amp_dev_upper);
8.      for (_dev_i = _dev_lower; _dev_i <= _dev_upper; _dev_i++) {
9.          for (_p_j = 1; _p_j < (m - 1); _p_j++) { /* replace with device variables, linearize 2-D array accesses */
10.             _p_resid = (((((ax * (_dev_uold[( _dev_i - 1) * 512 + _p_j] + _dev_uold[( _dev_i + 1) * 512 + _p_j])) \
11.                 + (ay * (_dev_uold[_dev_i * 512 + (_p_j - 1)] + _dev_uold[_dev_i * 512 + (_p_j + 1)]))) \
12.                 + (b * _dev_uold[_dev_i * 512 + _p_j])) - _dev_f[_dev_i * 512 + _p_j]) / b);
13.             _dev_u[_dev_i * 512 + _p_j] = (_dev_uold[_dev_i * 512 + _p_j] - (omega * _p_resid));
14.             _p_error = (_p_error + (_p_resid * _p_resid));
15.         } } /* thread block level reduction for float type*/
16.         xomp_inner_block_reduction_float(_p_error, _dev_per_block_error, 6);
17.     }
```

CPU Code: data management, exe configuration, launching, reduction

```
1.  xomp_deviceDataEnvironmentEnter(); /* Initialize a new data environment, push it to a stack */
2.  float *_dev_u = (float*) xomp_deviceDataEnvironmentGetInheritedVariable ((void*)u, _dev_u_size);
3.  /* If not inheritable, allocate and register the mapped variable */
4.  if (_dev_u == NULL)
5.  { _dev_u = ((float *) (xomp_deviceMalloc(_dev_u_size)));
6.    /* Register this mapped variable original address, device address, size, and a copy-back flag */
7.    xomp_deviceDataEnvironmentAddVariable ((void*)u, _dev_u_size, (void*) _dev_u, true);
8.  }
9.  /* Execution configuration: threads per block and total block numbers */
10. int _threads_per_block_ = xomp_get_maxThreadsPerBlock();
11. int _num_blocks_ = xomp_get_max1DBlock((n - 1) - 1);
12. /* Launch the CUDA kernel ... */
13. OUT__1__10117__<<<_num_blocks_,_threads_per_block_,(_threads_per_block_ * sizeof(float))>>> \
14.   (n,m,omega,ax,ay,b,_dev_per_block_error,_dev_u,_dev_f,_dev_uold);
15. error = xomp_beyond_block_reduction_float(_dev_per_block_error,_num_blocks_,6);
16. /* Copy back (optionally) and deallocate variables mapped within this environment, pop stack */
17. xomp_deviceDataEnvironmentExit();
```

Preliminary results: AXPY ($Y=a*X$)



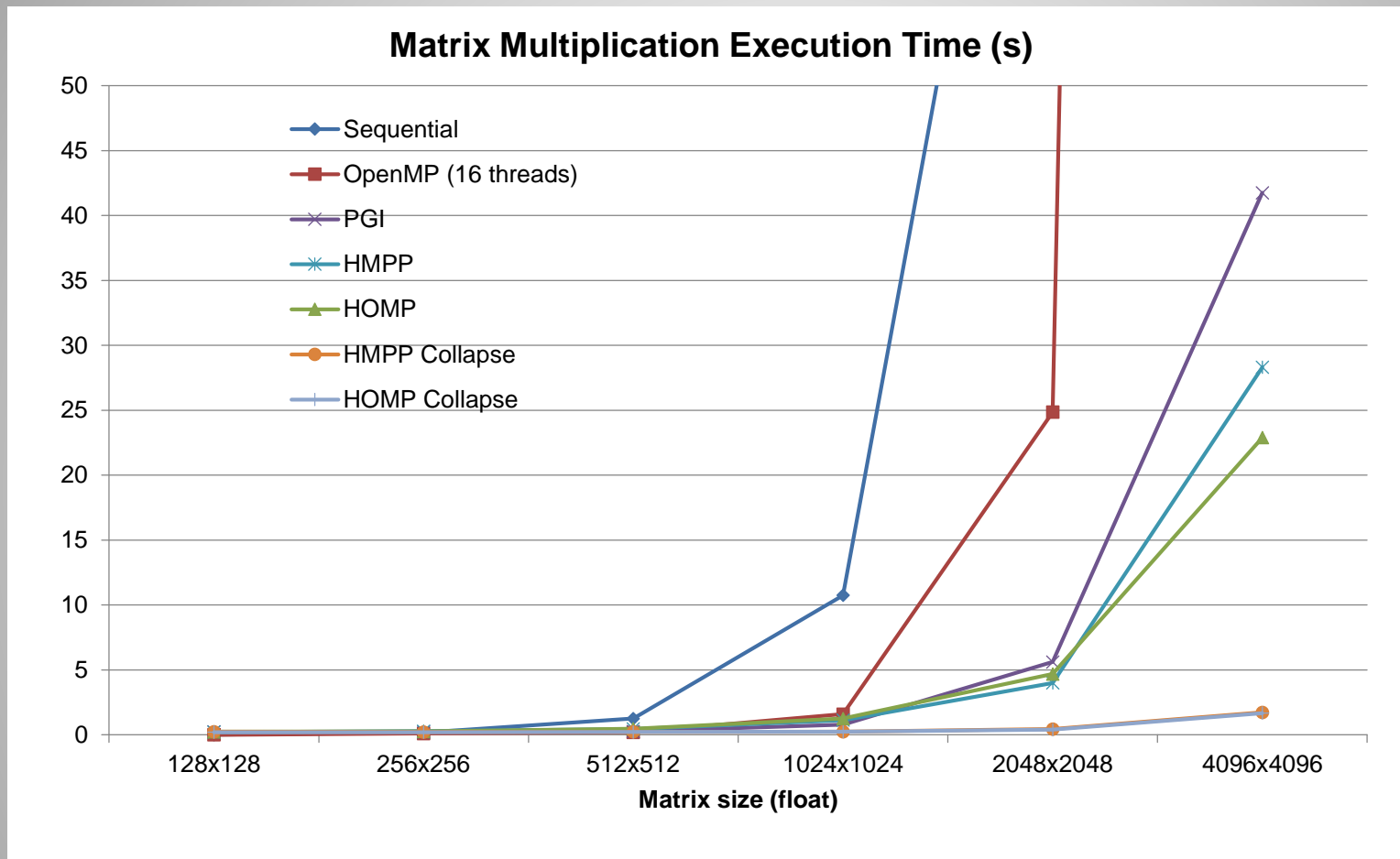
Hardware configuration:

- 4 quad-core Intel Xeon processors (16 cores) 2.27GHz with 32GB DRAM.
- NVIDIA Tesla K20c GPU (Kepler architecture)

Software configuration:

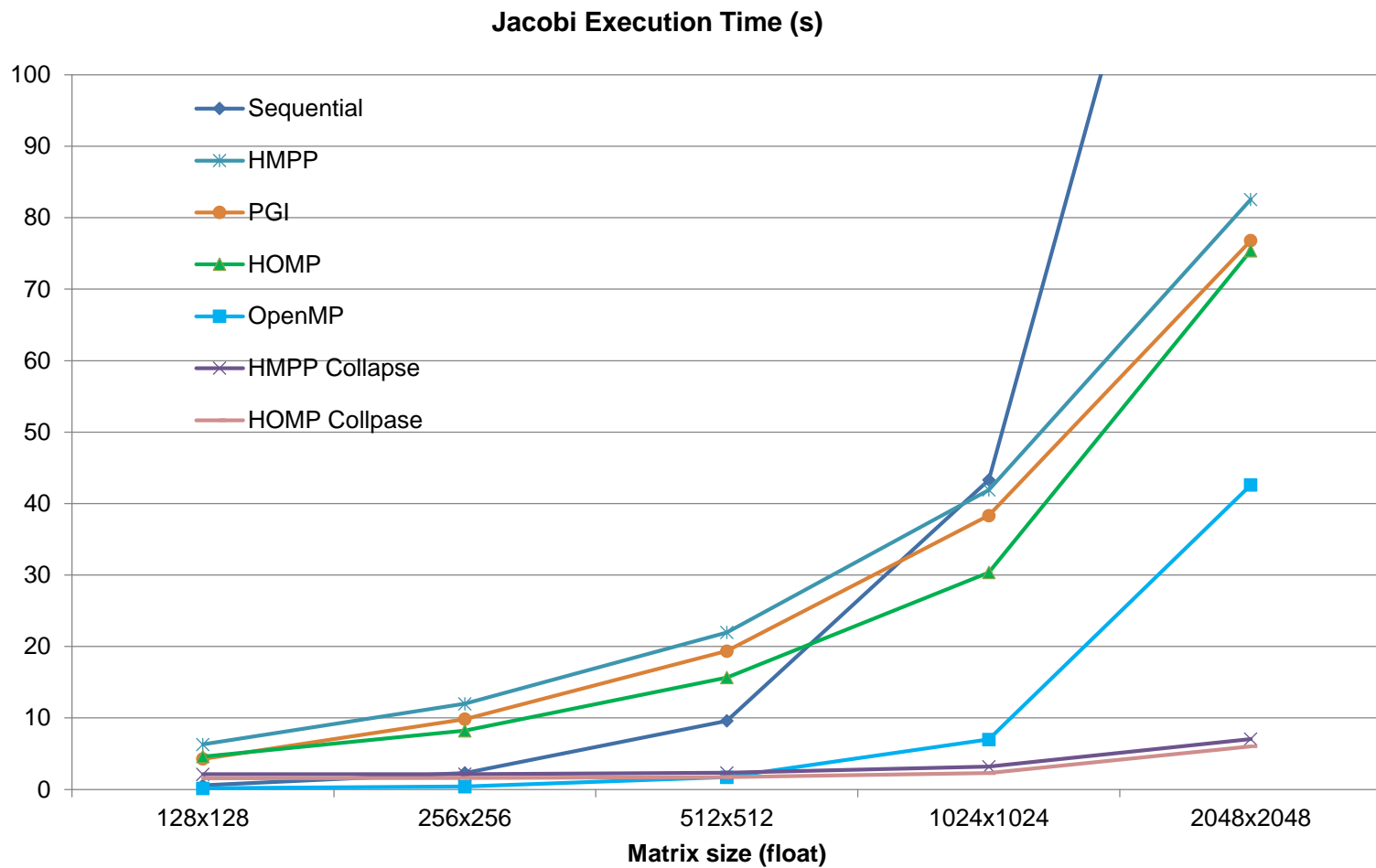
- HOMP (GCC 4.4.7 and the CUDA 5.0)
- PGI OpenACC compiler version 13.4
- HMPP OpenACC compiler version 3.3.3

Matrix multiplication

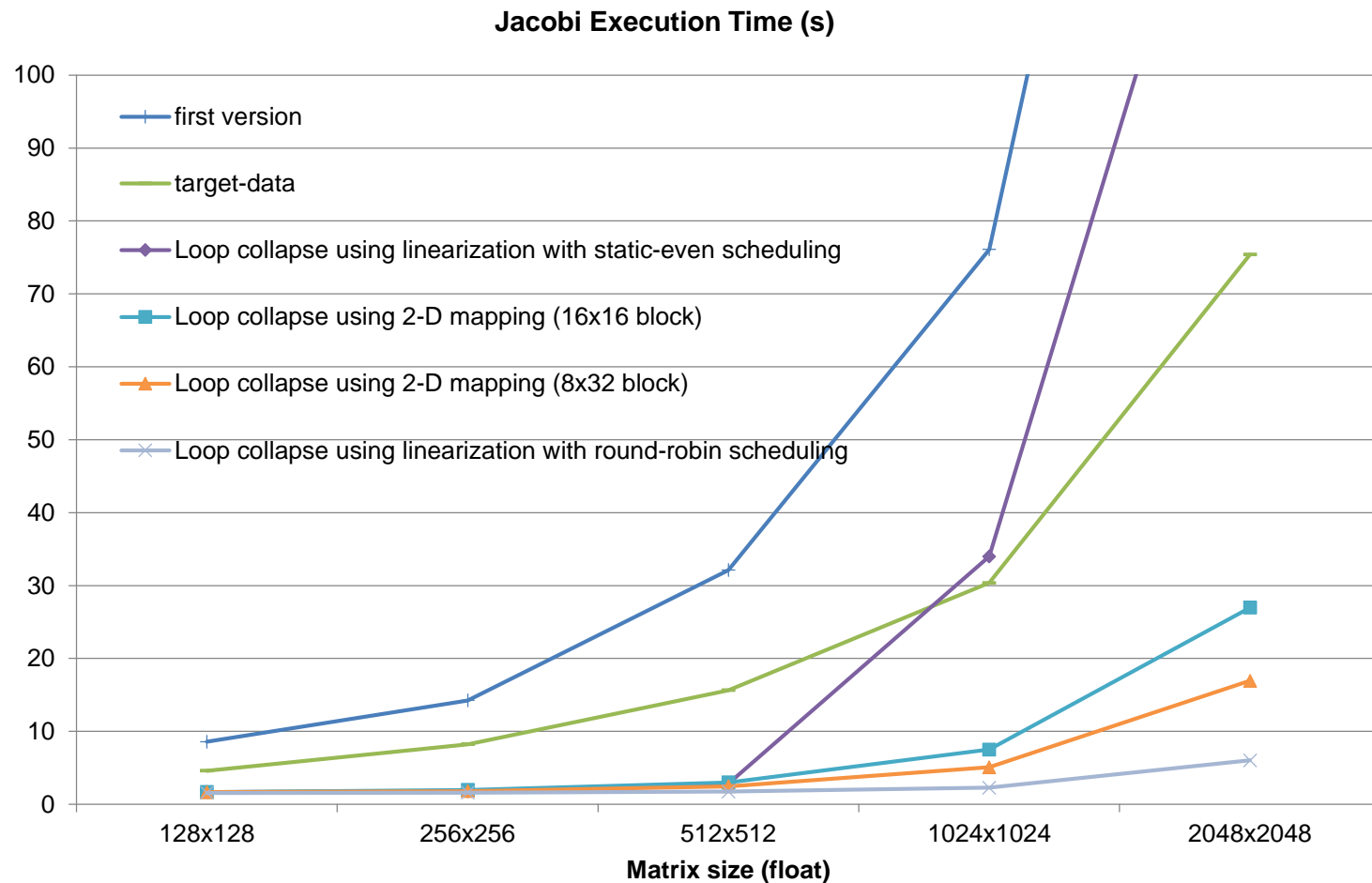


*HOMP-collapse slightly outperforms HMPP-collapse when linearization with round-robin scheduling is used.

Jacobi



HOMP: multiple versions of Jacobi



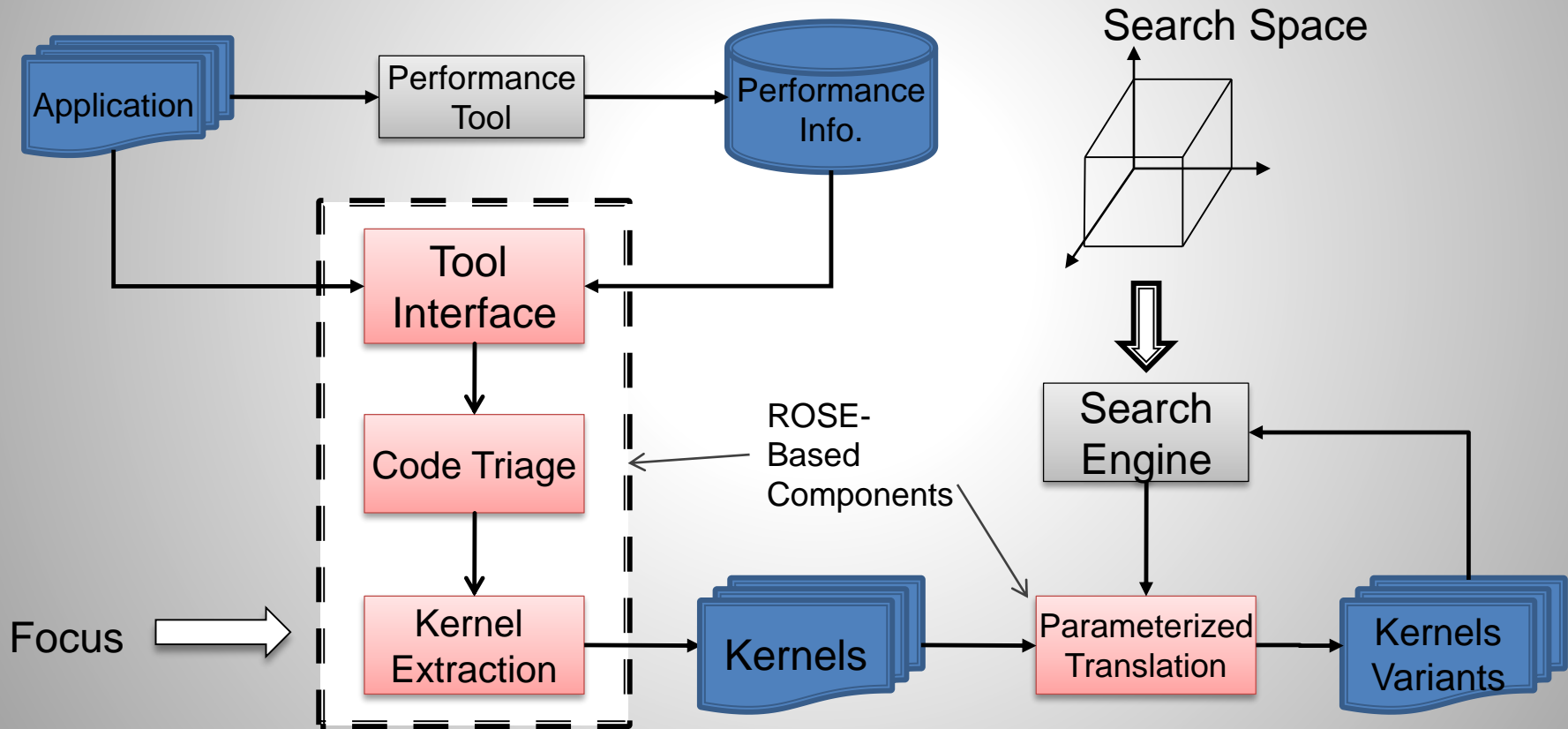
Case 1: conclusion

- Many fundamental source-to-source translation
 - CUDA kernel generation, loop transformation, data management, reduction, loop scheduling, etc.
 - Reusable across CPU and accelerator
- Quick & competitive results:
 - 1st implementation in the world in 2013
 - Interactions with IBM, TI and GCC compiler teams
 - Early Feedback to OpenMP committee
 - Multiple device support
 - Per device default loop scheduling policy
 - Productivity: combined directives
 - New clause: no-middle-sync
 - Global barrier: `#pragma omp barrier [league|team]`
 - Extensions to support multiple GPUs in 2015

Case 2: Performance via whole program autotuning

- Problem: increasingly difficult to conduct performance optimization for large scale applications
 - Hardware: multicore, GPU, FPGA, Cell, Blue Gene
 - Empirical optimization (autotuning):
 - small kernels, domain-specific libraries only
- Our objectives:
 - Research into autotuning for optimizing large scale applications
 - sequential and parallel
 - Develop tools to automate the life-cycle of autotuning
 - Outlining, parameterized transformation (loop tiling, unrolling)

An end-to-end whole program autotuning system



Kernel extraction: outlining

- Outlining:
 - Form a function from a code segment (outlining target)
 - Replace the code segment with a call to the function (outlined function)
- A critical step to the success of an end-to-end autotuning system
 - Feasibility:
 - whole-program tuning -> kernel tuning
 - Relevance
 - Ensure semantic correctness
 - Preserve performance characteristics

Example: Jacobi using a classic algorithm

```
int n,m;
double
u[MSIZE][MSIZE],f[MSIZE][MSIZE],uold[MSIZE][MSIZE];
void main()
{
    int i,j;
    double omega,error,resid,ax,ay,b;
    // initialization code omitted here
    // ...
    for (i=1;i<(n-1);i++)
        for (j=1;j<(m-1);j++) {
            resid = (ax * (uold[i-1][j] + uold[i+1][j]) + ay * (uold[i][j-1]
+ uold[i][j+1]) + b * uold[i][j] - f[i][j])/b;
            u[i][j] = uold[i][j] - omega * resid;
            error = error + resid*resid;
        }
    error = sqrt(error)/(n*m);
    // verification code omitted here ...
}
```

- Algorithm: all variables are made parameters, written variables used via pointer dereferencing
- Problem: 8 parameters & pointer dereferences for written variables in C
 - Disturb performance
 - Impede compiler analysis

```
// .... some code is omitted
void OUT__1__4027__(int *ip__, int *jp__, double omega,
double *errorp__, double *residp__, double ax, double ay,
double b)
{
    for (*ip__=1;*ip__<(n-1);(*ip__)++)
        for (*jp__=1;*jp__<(m-1);(*jp__)++)
        {
            *residp__ = (ax * (uold[*ip__-1][*jp__] +
uold[*ip__+1][*jp__]) + ay * (uold[*ip__][*jp__-1] +
uold[*ip__][*jp__+1]) + b * uold[*ip__][*jp__] -
f[*ip__][*jp__])/b;

            u[*ip__][*jp__] = uold[*ip__][*jp__] - omega * (*residp__);
            *errorp__ = *errorp__ + (*residp__) * (*residp__);
        }
}
void main()
{
    int i,j;
    double omega,error,resid,ax,ay,b;
    //...
    OUT__1__4027__(&i,&j,omega,&error,&resid,ax,ay,b);
    error = sqrt(error)/(n*m);
    //...
}
```

Our effective outlining algorithm

- Rely on source level code analysis
 - Scope information: global, local, or in-between
 - Liveness analysis: at a given point, if the value's value will be used in the future (transferred to future use)
 - Side-effect: variables being read or written
- Decide two things about parameters
 - Minimum number of parameters to be passed
 - Minimum pass-by-reference parameters

$\text{Parameters} = (\text{AllVars} - \text{InnerVars} - \text{GlobalVars} - \text{NamespaceVars}) \cap (\text{LiveInVars} \cup \text{LiveOutVars})$

$\text{PassByRefParameters} = \text{Parameters} \cap ((\text{ModifiedVars} \cap \text{LiveOutVars}) \cup \text{ArrayVars})$

Eliminating pointer dereferences

- Variables pass-by-reference handled by classic algorithms: pointer dereferences
- We use a novel method: variable cloning
 - Check if such a variable is used by address:
 - C: `&x`; C++: `T & y=x`; or `foo(x)` when `foo(T&)`
 - Use a clone variable if x is NOT used by address

$\text{CloneCandidates} = \text{PassByRefParameters} \cap \text{PointerDereferencedVars}$

$\text{CloneVars} = (\text{CloneCandidates} - \text{UseByAddressVars}) \cap \text{AssignableVars}$

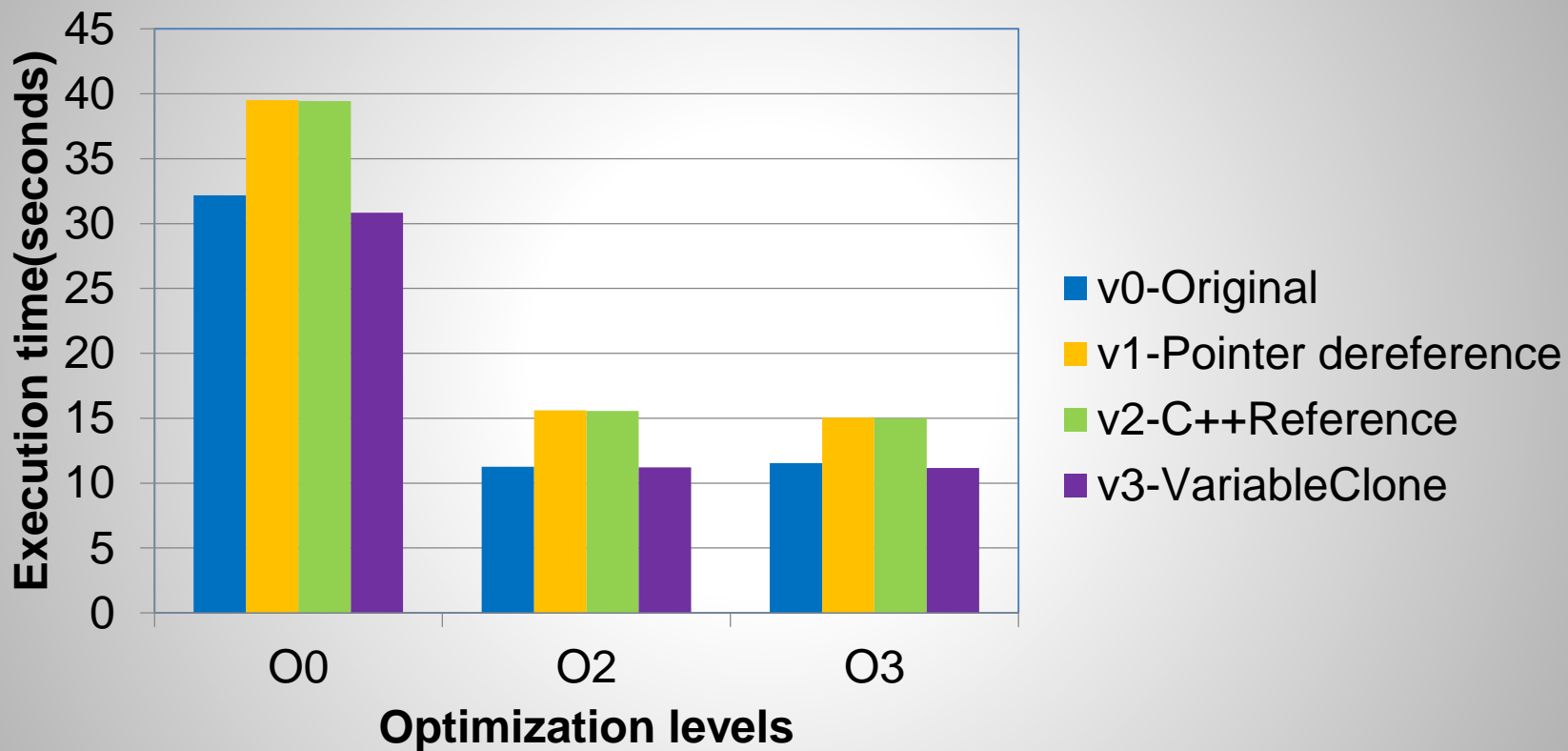
$\text{CloneVarsToInit} = \text{CloneVars} \cap \text{LiveInVars}$

$\text{CloneVarsToSave} = \text{CloneVars} \cap \text{LiveOutVars}$

Classic vs. effective outlining

Classic algorithm	Our effective algorithm
<pre> void OUT__1__4027__(int *ip, int *jp, double omega, double *errorp__, double *residp__, double ax, double ay, double b) { for (*ip__=1;*ip__<(n-1);(*ip__)++) for (*jp__=1;*jp__<(m-1);(*jp__)++) { *residp__ = (ax * (uold[*ip__-1][*jp__] + uold[*ip__+1][*jp__]) + ay * (uold[*ip__][*jp__-1] + uold[*ip__][*jp__+1]) + b * uold[*ip__][*jp__] - f[*ip__][*jp__])/b; u[*ip__][*jp__] = uold[*ip__][*jp__] - omega * (*residp__); *errorp__ = *errorp__ + (*residp__) * (*residp__); } } </pre>	<pre> void OUT__1__5058__(double omega,double *errorp__, double ax, double ay, double b) { int i, j; /* neither live-in nor live-out*/ double resid ; /* neither live-in nor live-out */ double error ; /* clone for a live-in and live-out parameter */ error = * errorp__; /* Initialize the clone*/ for (i = 1; i < (n - 1); i++) for (j = 1; j < (m - 1); j++) { resid = (ax * (uold[i - 1][j] + uold[i + 1][j]) + ay * (uold[i][j - 1] + uold[i][j + 1]) + b * uold[i][j] - f[i][j]) / b; u[i][j] = uold[i][j] - omega * resid; error = error + resid * resid; } *errorp__ = error; /* Save value of the clone*/ } </pre>

Different outlining methods: performance impact



Jacobi (500x500)

Whole program autotuning example

The Outlined SMG 2000 kernel (simplified version)

```
for (si = 0; si < stencil_size; si++)
  for (k = 0; k < hypr__mz; k++)
    for (j = 0; j < hypr__my; j++)
      for (i = 0; i < hypr__mx; i++)
        rp[ri + i + j * hypr__sy3 + k * hypr__sz3] -=
          Ap_0[i + j * hypr__sy1 + k * hypr__sz1 + A->data_indices[m][si]] *
          xp_0[i + j * hypr__sy2 + k * hypr__sz2 + dxp_s[si]];
```

- SMG (semicoarsing multigrid solver) 2000
 - 28k line C code, stencil computation
 - a kernel ~45% execution time for 120x120x129 data set
- HPCToolkit (Rice), the GCO search engine (UTK)

Results

Search space specifications			
Sequential		OpenMP	
Tiling size (k,j,i)	[0, 55, 5]	#thread	[1, 8, 1]
Interchange (all levels)	[0, 23, 1]	Schedule policy	[0, 3, 1]
Unrolling (innermost)	[0, 49, 1]	Chunk size	[0, 60, 2]

	icc -O2	Seq. (0,8,0)	OMP (6,0,0)
Kernel	18.6s	13.02s	3.35s
Kernel speedup	N/A	1.43x	5.55x
Whole app.	35.11s	29.71s	19.90s
Total speedup	N/A	1.18x	1.76x

- Sequential: ~40 hours for 14,400 points
 - Best point: (k,j,si,i) order
- OpenMP: ~ 3 hours for <992 points
 - Best point: 6-thread

Case 2 conclusion

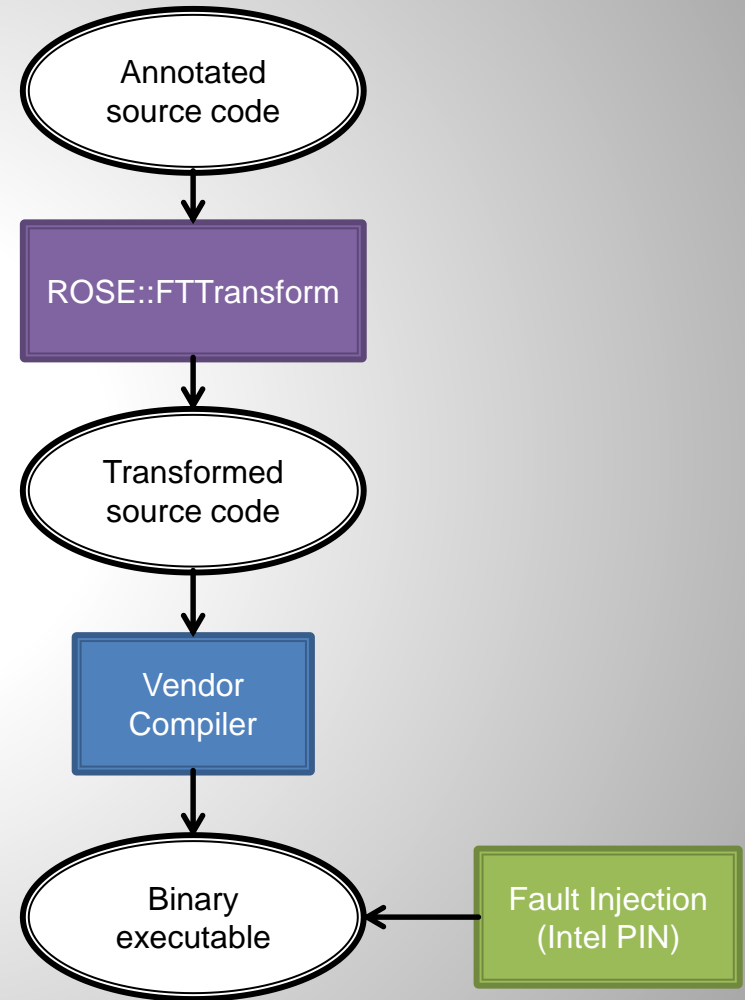
- Performance optimization via whole program tuning
 - Key components: outliner, parameterized transformation (skipped in this talk)
 - ROSE outliner: improved algorithm based on source code analysis
 - Preserve performance characteristics
 - Facilitate other tools to further process the kernels
- Difficult to automate
 - Code triage: The right scope of a kernel, specification
 - Search space: Optimization/configuration for each kernel

Case 3: resilience via source-level redundancy

- Exascale systems expected to use large-amount of components running at low-power
 - Exa- 10^{18} / 2GHz (10^9) per core → 10^9 core
 - Increased sensitivity to external events
 - Transient faults caused by external events
 - Faults which don't stop execution but silently corrupt results.
 - Compared to permanent faults
- Streamlined and simple processors
 - Cannot afford pure hardware-based resilience
 - Solution: software-implemented hardware fault tolerance

Approach: compiler-based transformations to add resilience

- Source code annotation(pragmas)
 - What to protect
 - What to do when things go wrong
- Source-to-source translator
 - Fault detection: Triple-Modular Redundancy (TMR)
 - Implement fault-handling policies
 - Vendor compiler: binary executable
 - Intel PIN: fault injection
- Challenges
 - Overhead
 - Vendor compiler optimizations



Naïve example

Policy	Final-wish
Pragma	<pre>#pragma FT-FW (Majority-voting) Y = ...;</pre>
Semantics	<pre>// T-Modular Redundancy y[0] = . . . ; y[1] = . . . ; y[2] = . . . ; // Fault detection if (!EQUALS(y[0] , y[1], y[2]))) { // Fault handling Y=Voting(y[0] , y[1], y[2]); } else Y = y[0];</pre>

Source code redundancy: reduce overhead

```
1  /* Original Jacobi 1-D , 3-points computation kernel */
2  void kernell()
3  {
4      int i;
5      for (i=1; i<SIZE-1; i=i+1)
6      {
7          d[i] = 0.25*c[i-1] + 0.5*c[i] + 0.25*c[i+1];
8      }
9  }
10 /* Transformed kernel with redundant computation */
11 void kernel2(double *c2)
12 {
13     double B_intra[3];
14     int i;
15     for (i=1; i<SIZE-1; i=i+1)
16     {
17         /* Baseline double modular redundancy (DMR) */
18         B_intra[0]= 0.25*c[i-1]+0.5*c[i]+ 0.25*c[i+1];
19         B_intra[1]= 0.25*c2[i-1]+0.5*c2[i]+ 0.25*c2[i+1];
20         d[i]= B_intra[0];
21         if (!equal(B_intra[0], B_intra[1], d[i]))
22         {
23             /* Additional N-2 redundancy and
24              * fault handling mechanism omitted here ... */
25         }
26     }
27 }
28 ...
29 /* call site doing pointer declaration and assignment */
30 double *c2 = c;
31 kernel2(c2);
```

Statement to be protected

Relying on baseline double modular redundancy (DMR) to help reduce overhead

Source code redundancy: bypass compiler optimizations

```
1  /* Original Jacobi 1-D , 3-points computation kernel */
2  void kernell ()
3  {
4      int i;
5      for (i=1; i<SIZE-1; i=i+1)
6      {
7          d[i] = 0.25*c[i-1] + 0.5*c[i] + 0.25*c[i+1];
8      }
9  }
10 /* Transformed kernel with redundant computation */
11 void kernel2(double *c2 )
12 {
13     double B_intra[3];
14     int i;
15     for (i=1; i<SIZE-1; i=i+1)
16     {
17         /* Baseline double modular redundancy (DMR) */
18         B_intra[0]= 0.25*c[i-1]+0.5*c[i]+ 0.25*c[i+1];
19         B_intra[1]= 0.25*c2[i-1]+0.5*c2[i]+ 0.25*c2[i+1];
20         d[i]= B_intra[0];
21         if (!equal(B_intra[0], B_intra[1], d[i] )
22             {
23             /* Additional N-2 redundancy and
24             fault handling mechanism omitted here... */
25             }
26     }
27 }
28 ...
29 /* call site doing pointer declaration and assignment */
30 double *c2 = c;
31 kernel2(c2);
```

Statement to be protected

Using an extra pointer to help preserve source code redundancy

Results: necessity and effectiveness of optimizer-proof code redundancy

- Hardware counter to check if redundant computation can survive compiler optimizations
 - PAPI (PAPI_FP_INS) to collect the number of floating point instructions for the protected kernel
 - GCC 4.3.4, O1 to O3

Transformation Method	PAPI_FP_INS (O1)	PAPI_FP_INS (O2)	PAPI_FP_INS (O3)
Our method of using pointers	Doubled	Doubled	Doubled
Naïve Duplication	No Change	No Change	No Change

Results: overhead

- Jacobi kernel:

- Overhead: 0% to 30%
- Minimum overhead
 - Stride=8
 - Array size 16Kx16K
 - Double precision
- The more original latency, the less overhead of added redundancy

- Livermore kernel

- Kernel 1 (Hydro fragment) – 20%
- Kernel 4 (Banded linear equations) – 40%
- Kernel 5 (Tri-diagonal elimination) – 26%
- Kernel 11 (First sum) – 2%

	1-D 1-Point	1-D 3-Point	2-D 5-Point
Iteration stride = 1			
Array size: 1 million for 1-D, 4096x4096 for 2-D			
float	17.33%	29.91%	30.19%
double	27.55%	22.27%	22.60%
Array size: 16 million for 1-D, 16Kx16K for 2-D			
float	13.87%	25.58%	25.03%
double	17.43%	19.97%	17.37%
Iteration stride = 8			
Array size: 1 million for 1-D, 4096x4096 for 2-D			
float	8.36%	19.12%	25.39%
double	5.10%	6.33%	5.44%
Array size: 16 million for 1-D, 16Kx16K for 2-D			
float	3.57%	10.30%	14.54%
double	0.05%	0.80%	1.59%

Conclusion

- HPC challenges
 - Traditional: performance, productivity, portability
 - Newer: Heterogeneity, resilience, power efficiency
 - Made much worse by increasing complexity of computation node architectures
- Source-to-source compilation techniques
 - Complement traditional compilers aimed for generating machine code
 - Source code analysis, transformation, optimizations, generation, ...
 - High-level Programming models for GPUs
 - Performance tuning: code extraction and parameterized code translation/generation
 - Resilience via source level redundancy

Future directions

- Accelerator optimizations
 - Newer accelerator features: Direct GPU to GPU communication, Peer-to-Peer execution model, Uniform Virtual Memory
 - CPU threads vs. accelerator threads vs. vectorization
- Knowledge-driven compilation and runtime adaptation
 - Explicit represent optimization knowledge and automatic apply them in HPC
- A uniform directive-based programming model
 - A single X instead of MPI+X to cover both on-node and across node parallelism
- Thread level resilience
 - Most resilience work focus on MPI, not threading