HERCULES: A PATTERN DRIVEN CODE TRANSFORMATION SYSTEM

Christos Kartsaklis
Oscar Hernandez
Chung-Hsing Hsu
Thomas Ilsche
Wayne Joubert
Richard Graham

HIPS 2012 (May 21, 2012)
Take Home Message

User-oriented tools are more important than ever for programming the new leadership class supercomputers
New Leadership Class Machines

- **Titan**: 10-30PF Cray XK6 (ORNL)
  - Accelerator based

- **Path Forward to Exascale**:
  - 100,000,000 order cores
  - Extreme levels of parallelism
    - Thread-based programming
    - Task-based execution models
  - Either on die or off die
  - Lower memory footprint per core
  - Deeper memory hierarchies
  - Component failure is the norm

```
<table>
<thead>
<tr>
<th></th>
<th>2020</th>
<th>2018</th>
<th>Factor Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>System peak</td>
<td>2 PF/s</td>
<td>1 EF/s</td>
<td>500</td>
</tr>
<tr>
<td>Power</td>
<td>6 MW</td>
<td>20 MW</td>
<td>3</td>
</tr>
<tr>
<td>System Memory</td>
<td>0.3 PB</td>
<td>10 PB</td>
<td>33</td>
</tr>
<tr>
<td>Node Performance</td>
<td>0.125 GF/s</td>
<td>10 TF/s</td>
<td>80</td>
</tr>
<tr>
<td>Node Memory BW</td>
<td>25 GB/s</td>
<td>400 GB/s</td>
<td>16</td>
</tr>
<tr>
<td>Node Concurrency</td>
<td>12 CPUs</td>
<td>1,000 CPUs</td>
<td>83</td>
</tr>
<tr>
<td>Interconnect BW</td>
<td>1.5 GB/s</td>
<td>50 GB/s</td>
<td>33</td>
</tr>
<tr>
<td>System Site (nodes)</td>
<td>20 K nodes</td>
<td>1 M nodes</td>
<td>50</td>
</tr>
<tr>
<td>Total Concurrency</td>
<td>225 K</td>
<td>1 B</td>
<td>4,444</td>
</tr>
<tr>
<td>Storage</td>
<td>15 PB</td>
<td>300 PB</td>
<td>20</td>
</tr>
<tr>
<td>Input/Output bandwidth</td>
<td>0.3 TB/s</td>
<td>20 TB/s</td>
<td>100</td>
</tr>
</tbody>
</table>
```
Programming Challenges

• Optimization strategies become complex
  – High-levels of concurrency, complex analyses (inter-procedural dependences, scoping of data, global data usage)

• Significant restructuring of applications to new platforms.
  – Significant amount of time consuming, time to invest vs. profitability with little reuse of knowledge.
  – Application tied to architecture-specific optimizations

• Multiple programming models and languages.
  – How to map discovered parallelism to programming model, then architecture.
    • What should run on cores, GPUs, across nodes? Load Balance vs. Locality
  – Each with different optimization strategies

• Constant adaptation to new architectures
  – Efficient translation of code to architecture
Challenges (Cont..)

- Meta programming source-to-source translators rely on backend compilers to generate efficient code:
  - Procedure Inlining
  - Constant Propagation
  - Dead code elimination
  - Common sub-expression elimination
  - Loop optimizations with STL iterators
  - Data flow analysis
  - Alias Analysis

- Programmers may have to write code in assembler if back-end compiler doesn’t generate efficient code.

```cpp
template <class T>
struct AxyOp {
    const T * x;
    T * y;
    T alpha, beta;
    void execute(int i)
    { y[i] = alpha*x[i] + beta*y[i]; }
};

AxyOp<double> op;
op.x = ...; op.alpha = ...;
op.y = ...; op.beta = ...;
node.parallel_for< AxyOp<double> >
    (0, length, op);
```
Manage the complexity for the user

Assist the user in different use-case scenarios:

• Find any nested loop that expands inter-procedurally, where the two outmost loops are parallelizable and inner loops vectorizable with multiply adds.
  – => Parallelize outer loop with OpenMP (auto-scope)
  – => Parallelize second loop with Accelerator Directives (Grid)
  – => Parallelize vector loops with Accelerator Directives (Threadblocks)

• Find vectorizable loops with non-contiguous data accesses and a consumed volume of data less than X

• Find data accesses of a structure inter-procedurally, with no pointers, and that is accessed in loopnests.
  – Re-layout data structure
Talk Outline

• Related Work
• HERCULES
• Lesson Learned
Many Stages in Translation

User

Translation Tools

Optimized Application

Compilers

Application Binary / Libraries

Execution in Exascale/Petascale Systems

Data Collection

Performance Analysis Auto-Tuning

Visualization, Feedback to User

User
Related work

• User-driven source-to-source translators
  – Poet, Loop Optimizer: Rose (C++/C/Fortran, OpenMP, UPC)
  – ChiLL: Suif (C support)
  – Orio/PluTo: (C/OpenMP)
  – TSF: Forsys (Fortran)

• Loose integration between translation tool and back-end compiler.
  – Based on parsers that operate at the abstract syntax tree level, with limited analysis and target-specific transformations.
  – Rely on back-end compiler optimizations

• Limited feedback from the translation tool to the user.

• Strategies tied to a particular user’s source code
HERCULES

• A pattern driven code translation tool

• Distinctive features:
  – Infrastructure to manage program analysis information for the user to facilitate the understanding of the application
  – Automates the process of applying transformations multiple times throughout the code base
  – Principle of separation of concerns: Application Science vs. Optimizations
  – Documents the transformation process done by computational scientists
  – Reusability of transformation workflow
  – Works with an underlying compiler infrastructure and is a solution that is implementable in compilers.

Helps to capture the application needs
Implemented HERCULES Architecture

Application → Facts, IR, Analysis Extractor → Program Database (Facts, Analysis, Representation)

Pattern ‘Query’ Action ‘Transformation’ → Translator

Translator → Pattern Matching Engine, Constraint-Based System

Feedback → Performance Analysis (Vampir)

Analyses → Source-to-Source Transformation → Compiler Plugin

Open64 | NVIDIA | Pathscale

Object Code

Transformations
Accomplishments

• Working system that can input patterns and transformation scripts and output transformed code and/or binaries.

• Implemented a compiler plug-in infrastructure to enable pattern-matching and transformations at different phases.

• Designed a pattern language, and built a directives parser and pattern-matching engine that uses underlying PROLOG technology.

• Progress on output program analysis to the program database
  – Parallelization information, access vectors, cost-models

• Implemented APIs for IR traversal and applying transformations:
  – Loop-transformations, specialization, instrumentation, data transformations, exposed more APIs.

• Worked with applications for requirements of pattern and transformation
  – CAM/SE, Sweep3D, S3D, HPL
Queries: Pattern Language Definition

- Incremental approach
  - Define using directives and source code language (C, C++, Fortran)
  - Patterns can be generalized incrementally
    - Can easily be used to match a specific source code to be transformed.
- Support for convenience functions
- Can be used to query for analyses
- Pattern can consist of syntactic properties of the code
  - Can be easily extended to support analyses and runtime characteristics of code

Pattern Language Definition:

```plaintext
#pragma hercules declare pattern (pattern name, return type)
#pragma hercules symbol bind([variable names],)
#pragma hercules statement bind
#pragma hercules insert ...
#pragma hercules use pattern_name (args)
#pragma hercules bind promote(expre).
```
Extraction and Pattern Matching Engine

- We translate the program to PROLOG to generate program facts.
  - Mappings between PROLOG facts and compiler intermediate representations are recorded

- Program analysis is also translated to PROLOG

- Pattern translates to PROLOG constraints to be solved with program facts.

```
#define N 100
int main() {
    int a[N], b[N], i
    for(i=0; i<N; i++)
        a[i] = b[i] + i;
    return a[50]
}
```

Program:
```
# define N 100
int main() {
    int a[N], b[N], i
    for(i=0; i<N; i++)
        a[i] = b[i] + i;
    return a[50]
}
```

Intermediate Representation

PROLOG Facts
```
(~12x)
AST_node(1,0,func_entry,func_entry).
AST_root_of(1,1).
AST_rtype(1,v).
AST_desc(1,v).
AST_kids_of(1,12,[2,3,1,4,5])
AST_has_st(1,12801).
ST_st_idx_to_st(12801,1,50,'main').
ST_has_sym(1,12801).
ST_index(50,class_func,text).
......
```
Compiler Plug-in API

- Implemented a unified compiler plug-in.

```c
typedef enum {
    PLUGIN_REQ_START = 0,
    PLUGIN_REQ_REGISTER = 1,
    PLUGIN_REQ_UNREGISTER = 2,
    PLUGIN_REQ_STATE = 3,
    PLUGIN_REQ_CURRENT_PHASE = 4,
    PLUGIN_REQ_STOP = 5,
    PLUGIN_REQ_PAUSE = 6,
    PLUGIN_REQ_RESUME = 7,
    PLUGIN_REQ_LAST = 8
} PLUGIN_API_REQUEST;

typedef enum {
    PLUGIN_EVENT_IPL_BEFORE = 1,
    PLUGIN_EVENT_IPL_AFTER = 2,
    PLUGIN_EVENT_IPA_BEFORE = 3,
    PLUGIN_EVENT_IPA_AFTER = 4,
    PLUGIN_EVENT_VHO_BEFORE = 5,
    PLUGIN_EVENT_VHO_AFTER = 6,
    PLUGIN_EVENT_LNO_BEFORE = 7,
    PLUGIN_EVENT_LNO_AFTER = 8,
    PLUGIN_EVENT_WOPT_BEFORE = 9,
    PLUGIN_EVENT_WOPT_AFTER = 10,
    PLUGIN_EVENT(CG) BEFORE = 11,
    PLUGIN_EVENT(CG) AFTER = 12,
    PLUGIN_EVENT_RESERVED = 13,
    PLUGIN_EVENT_LAST = 14
} PLUGIN_API_EVENT;

class plugin {
    bool init();
    void start();
    void stop();
    void pause();
    void resume();
    void register_event(PLUGIN_API_EVENT e, void (*func)(PLUGIN_API_EVENT e))
} ...
```

Lib Plugin.so

Compiler Plugin

Compilation Phase

Open64
NVIDIA
GCC

init()

register(...), start()

callback(event)

Invoke Transformation Script
HERCULES AST & Transformation APIs

Class HERCULES_AST {
  ....
  void Root();
  void Next();
  void Previous();
  void Set_Node();
  void Kid(INT i);
  char * Get_Symbol_Name();
  INT Num_Kids();
  OPERATOR Operator();
  void Statements();
  void This_Tree();
  void Symbol();
  void Type();
  void Find();
  void Find_Symbols();
  void Find_Operator();
  void Unroll();
  void Specialize();
  void Instrument();
  void Parent();
  Transformations();
};

void MatrixVectorMultiply(double *C, double *A, double *B, int dimension) {
  int i, j;
  for(i=0 ; i<dimension ; i++)
  {
    C[i] = 0.0;
    for(j=0 ; j<dimension ; j++)
      C[i] = C[i] + A[i*dimension+j]*B[j];
  }
}

void Root();
Kid(5); Next()
Kid(1); Operator();
Example of a Simple Hercules Pattern

```c
void mypattern_driver() {
    #pragma hercules pattern declare mpi_loop_pattern (statement FOR, list : statement LEPOINTS)
    #pragma hercules symbol expr1 promote(expression)
    int expr1;
    #pragma hercules statement insert ...
    #pragma hercules symbol i bind(I)
    int i=0;
    #pragma hercules statement bind(FOR) exit_points(LEPOINTS)
    for ( ; i<expr1 ; i++) {
        #pragma hercules statement insert ...
        #pragma hercules pattern use hspl_mpi_callsite(C)
        #pragma hercules statement insert ...
    }
    ....
}
```

Find all loop-nests, each of which contains an MPI call site, return all exit points.

Example of matched code:
```c
void foo(int b, int a) {
    int i=0;
    for ( ; i<b ; i++) {
        label1:
        MPI_Send();
        if (a) {
            goto label2;
        } else {
            goto label1;
        }
        if (MPI_Send()) { return; }
    }
    label2:
    bar();
}
```

Useful for instrumentation transformation or Performance analysis.
HERCULES in CAM/SE

- Pattern-based parallelization support to reuse transformation logic.

**PATTERN DEFINITION:**

```hercules
!$hercules pattern declare implicit_camse(statement TARGET)
do ie=nets, nete
!$hercules statement bind TARGET
do q=1,qsize
do k=1,nlev
do j=1,nv
do l=1,nv
!$hercules statement insert ...
do i=1,nv
!$hercules statement insert ...
end do
divp4da(l,j,k,q,ie)= rmetdetp(l,j,ie) * ((rdx(ie))*dudx00 ...) 
end do
!$hercules statement insert ...
end do
!$hercules statement insert ...
end do
```

**MATCHED AND TRANSFORMED CODE**

```fortran
do ie=nets, nete
!$omp parallel do private(K, Q, J, DUDX00, L, DUDY00I, I),
!$& shared(DVV, METDET, DINV, GRADQ5DA, RMETDETP,
!$& RDX, RDY, IE, DIVDP4DA)
do q=1,qsize
do k=1,nlev
do j=1,nv
do l=1,nv
dudx00=0.0d0
dudy00i=0.0d0
do i=1,nv
dudx00 = dudx00 + Dvv(i,I ) * (metdet(i,j,ie) ..
dudy00i = dudy00i + Dvv(i,j ) * (metdet(l,i,ie)) ..
end do
divp4da(l,j,k,q,ie)= rmetdetp(l,j,ie) * ((rdx(ie))*dudx00 ...) 
end do
end do
```

**TRANSFORMATION RECIPE:**

```bash
hercules_transformation_for(implicit_camse,1):=
hercules_invoke(s2s_omp_parallelization, arg0)
```

Transformed

Parallelization support and autoscooping of variables
HERCULES in 3D Sweep

Pattern Definition:
void sub1( double* v, double* w, double* matrix1,double* matrix2, double* sX, ... ) {  ....
#pragma hercules pattern declare specialized_pattern(statement FOR, this)
#pragma hercules symbol bind(IE, IU1, IU2, IM1, IM2)
#pragma hercules statement bind ... 
#pragma hercules statement bind(FOR)
for ( int IE=0 ; IE<nE; IE++ ) {
  for ( int iA=0; iA<nA; iA++ ) {
    //---First matvec.
    for ( int iU1=0; iU1<nU; iU1++ ) {
      SX(iU1) = 0.;
      for ( int iM1=0; iM1<nM; iM1++ ) {
        SX(iU1) += Matrix1(iA,iM1) * V(iE,iU1,iM1);
      }
    }
  }
} 
#pragma hercules statement insert ...

#pragma hercules statement bind ... 
for ( int iU2=0; iU2<nU; iU2++ ) {
  for ( int iM2=0; iM2<nM; iM2++ ) {
    if ( iA == 0 ) {
      W(iE,iU2,iM2) = 0.;
    }
    W(iE,iU2,iM2) += Matrix2(iA,iM2) * SX(iU2);
  }
} 
#pragma hercules statement insert ...

Transformation Recipe

__inline void sub1( ....)
{
  // specialized code to nU=4 and nM=16
  if((nU == 4) & (nM == 16))
  {
    for(iE0 = 0; iE0 <= (nE - 1); iE0 = iE0 + 1)
    {
      for(iA0 = 0; iA0 <= (nA - 1); iA0 = iA0 + 1)
      {
        if((iA0 == 8) // specialized code to iA0=8
        {
          (sX)[0] = 0.0;  // unrolled loop as a result of specialization.
        }
      }
    }
  }
  //---First matvec.
  for ( int iU1=0; iU1<nU; iU1++ ) {
    (sX)[0] = 0.0; // unrolled loop as a result of specialization.
    for ( int iM1=0; iM1<nM; iM1++ ) {
      (sX)[0] += Matrix1(iA,iM1) * V(iE,iU1,iM1);
    }
  }
  //---Second matvec.
  for ( int iU2=0; iU2<nU; iU2++ ) {
    for ( int iM2=0; iM2<nM; iM2++ ) {
      if(iA == (0)) {
        (w)[iE0 + (nE * (i + (iM0 * (4)))))] = 0.0;
      }
    }
  }
  //---Third matvec.
  for ( int iU2=0; iU2<nU; iU2++ ) {
    for ( int iM2=0; iM2<nM; iM2++ ) {
      if(iA == (0)) {
        (w)[iE0 + (nE * (i + (iM0 * (4)))))] = (w)[iE0 + (nE * (i + (iM0 * (4))))) +
          ((matrix2)((iA0 + (nA * iM0))) * (sX)[i]);
      }
    }
  }
  else
  {
    // non-specialized code
  }
}
Lessons Learned

• Constrained by the choice of the compiler
  – Compiler imposes an order/idiom for analysis/transformation.
  – Lowering and Normalization.

• Influenced by the choice of the pattern language
  – Syntax-directed may be limited.
  – Generalization versus specialization.
THANK YOU

This research was sponsored by the LDRD program of ORNL, managed by UT-Battelle, LLC for the DOE under Contract DE-AC05-00OR22725