CIS 631
Advanced Parallel Computing
Parallel Performance Analysis Environments

Prof. Allen D. Malony
Department of Computer and Information Science
Winter 2022
Logistics

- Problems with the AXIS GPUs
  - BIOS is not recognizing them
  - 64-bit base address register (BAR) issue
  - Still investigating …
- Will use Talapas instead
  - Slurm partition “cis631” containing a RACS dev node with 2 NVIDIA A100 GPUs
    - available 24x7
    - CIS 631 jobs will have dedicated access
  - Slurm reservations that reserve 2 general GPU nodes
    - total of 8 NVIDIA K80 GPUs
    - 25 hours each week dedicated for CIS 631 users
  - CIS 631 users may also access the general GPU node partitions
    - approximately 96 NVIDIA K80 GPUs
    - shared with research users
Tools / Technology Integration

Tools / Technologies
- TAU
- mpiP
- PAPI
- GPTL
- RCRToolkit
- PBound
- Roofline
- PEBIL
- PSiNtracer
- ROSE
- CHiLL
- Active
- Harmony
- Orio

Research Areas
- Performance
- Modeling
- Reliability
- Code analysis
- Autotuning

Integration End-to-end

Performance Optimization
SciDAC applications
Resilience
Energy

SUPER was a 5-year SciDAC Institute (2011-2016)
Working with the RAPIDS / RAPIDS-2 SciDAC Institute (2017-now)

CIS 631: Advanced Parallel Computing
SUPER focuses on integrating developed tools and technologies to build enhanced capabilities.
End-to-end Performance Optimization

- SUPER established processes for applying integrated tools for end-to-end optimization
SUPER Autotuning with Tools Integration

- SUPER applied autotuning to optimize HPC applications
  - Active Harmony autotuning system (Hollingsworth, UMD)
    - software architecture for optimization and adaptation
  - CHiLL compiler framework (Hall, Utah)
    - CPU and GPU code transformations for optimization
  - Orio annotation-based autotuning (Norris, UO)
    - code transformation with optimization (CUDA, OpenCL)

- Integrate performance tools (TAU) with these frameworks
  - Use to gather performance data for autotuning/specialization
  - Store performance data with metadata for each experiment variant and store in performance database (TAUdb)
  - Use machine learning and data mining to increase the level of automation of autotuning and specialization
TAU and Autotuning in SUPER

CHiLL + AH

Outline Function
- CHiLL
  - CHiLL Recipes
  - Selective Instrumentation File (specifying parameters to capture)
  - Instrumented Variant
  - Parameterized Performance Profile
- tau_instrumentor
- Search Driver (brute force or Active Harmony)

Instrumented Variant
- execute
- TAUdb

Selective Instrumentation File (specifying parameters to capture)
- Instrumented Variant
- Performed Instrumentation
- TAUdb

Parameterized Performance Profile
- execute
- TAUdb

ROSE

PerfilExplorer

TAUdb

CUDA

OpenCL

TAUdb

TAU Profiles

Linked at Runtime

CUPTI callback measurement library

PAPI

Orio Code Generator

Experiment

TAU Metadata Entries

Transformations

Execution Time

CUDA

MPAS-O

MPAS

XGC1

Geant4

CESM
**CHiLL: Compiler-based Autotuning**

- CHiLL is a compiler framework for loop transformations based on the polyhedral model.
- Used to automatically generate specialized versions according to specified optimization strategies.
  - CHiLL provides a high-level scripting interface.
  - Allows a user or a compiler to specify a sequence of code transformations and parameters.
  - Describes the search space of specialized implementations.
  - Heuristics trim the search space.
- Variants are generated, measured, and analyzed.
CHiLL Transformation Scripts / Parameters

- Transformation scripts are parameterized

- Consider matrix multiply

```c
for(j=0; j < n; j++) {
    for(k=0; k < n; k++) {
        for(i=0; i < n; i++) {
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
        }
    }
}
```

- TJ, TI, TK determine tile sizes

- UI, UJ determine unroll factors

- Overall search space is \{parameters\} x \{scripts\}:
  - Each is a transformation that produces a code variant
Autotuning Integration (CHiLL + AH)

Process
- search
- transformation
- measurement
- storage
- feedback

Measurement
- Parameter profiling
- TAUdb storage

CHiLL Recipes

Search Driver (brute force or Active Harmony)

CHiLL

code variant

Selective Instrumentation File (specifying parameters to capture)

tau_instrumentor

Instrumented Variant

execute

Parameterized Performance Profile

tau_instrumentor

ROSE outliner

Outlined Function

Selective Instrumentation File (specifying parameters to capture)

tau_instrumentor

Instrumented Function

execute

Parameterized Performance Profile

TauDB
Autotuning with TauDB Methodology

- Each time the program executes a code variant, we store metadata in the performance database indicating by what process the variant was produced:
  - Source function
  - Name of CHiLL recipe
  - Parameters to CHiLL recipe

- The database also contains metadata on what parameters were called and also on the execution environment:
  - OS name, version, release, native architecture
  - CPU vendor, ID, clock speed, cache sizes, # cores
  - Memory size
  - Any metadata specified by the end user
Learning Performance Specialization

- Apply machine learning to data stored in TAUdb
  - Generate decision trees based upon code features

```c
void matmult(float **c, float **a, float **b, int L, int M, int N)
```

parameterize using L, M, N
Decision Code Generation to Effect Specialization

- Use a ROSE-based tool to generate a wrapper function
  - Carries out the decisions in the decision tree and executes the best code variant

- Decision tree code generation tool takes Weka-generated decision tree and decision functions
  - If using custom metadata, user provides a custom decision function
  - Decision functions for metadata automatically collected by TAU
Orio: DSL-based Autotuning Framework

- Orio attempts to express any properties of the computation that can be exploited to optimize performance

Approach
- Define new DSLs or embeddable restricted GPLs
- Augmentable with optimization specifications
  - typical optimizations (e.g., tiling, unrolling, …)
  - specialized versions (e.g., different input sizes, …)
- Transform code based on knowledge in various forms
- Evaluate variant performance (different code output)
- Search for best
Orio Empirical-Based Autotuning Process

Code with DSL Annotations → DSL Parser → Sequence of (Nested) Annotated Regions

Transformed Code

Code Generator → Code Transformations

 Empirical Performance Evaluation

Search Engine

best performing version

Optimized Code

CUDA, OpenCL, Fortran, C
Orio and TAU Integration

- Orio Code Generator
- OrCUDA
- OrCL
- Experiment
- OpenCL
- TAU OpenCL Library Wrapper
- CUPTI callback measurement library
- TAU Metadata Entries
  - Transformations
  - Execution Time
- TAU Profiles
- TAUdb
- Measurements
  - Metric profiling
  - Metadata
  - TAUdb storage

- Links at Runtime
- Uploads
- Autotuning analysis
  - Machine learning
  - Optimization search
  - Specialization
Orio and TAU Integration

- Source code annotations (pragmas) allow Orio to generate a set of low-level performance optimizations
  - Focus on code for accelerator cards
  - Run kernel after each transformation applied
  - Set of optimizations searched to find best for kernel

- Integrated TAU with Orio to collect performance data and metadata about each experiment that Orio runs
  - Metadata describes the applied transformations and environment (accelerator, memory sizes, …)

- Stored in TAUdb and are fed back to Orio

- Allows Orio to tune based on any TAU metric
Orio Tuning of Vector Multiplication

- Orio tuning of a simple 3D vector multiplication
- 2,048 experiments fed into TAUdb
  - Use PerfExplorer with Weka to do component analysis

<table>
<thead>
<tr>
<th>Threads Per Block</th>
<th># of Blocks</th>
<th>Unroll factor</th>
<th>Preferred L1 Size</th>
<th>CFLAG</th>
<th>Warps Per SM</th>
</tr>
</thead>
</table>

![Graphs showing Kernel Execution Time, GPU occupancy, and Number of Threads vs. Threads Per Block and # of Blocks.](image_url)
Orio Enables Multi-Target Autotuning

- Focus on accelerators
- Extended CUDA support with OpenCL
  - OrCL
    - OpenCL code generation with annotations based on tunable OpenCL parameters
- Allows for tuning experiments to be conducted on a broader set of accelerator platforms
- Autotuning experiments
  - BLAS kernels
  - Radiation transport code
Autotuning

Definition
- Automatically generate a search space of possible implementations of a computation
  - Code variant: unique implementation of a computation
  - Parameter: discrete set of values that govern code generation or execution of a variant
  - Measure: execution time and compare
  - Select: the best-performing implementation

Identifying and pruning the search space is key

Three types of autotuning
- Library-based: ATLAS, FFTW, …
- Application-based: PetaBricks, Active Harmony, …
- Compiler-based: CHiLL/AH, Orio, …
Autotuning Methodology
Accelerators

- Devices that allow offloading of work from the CPU onto a specialized coprocessor
- GPUs
  - Massive number of very simple cores
- Intel Xeon Phi
  - Moderate number of moderate-complexity cores
- FPGA
  - Programmable hardware
- How do we program these devices
Our Motivation Criteria

- Ideally, we want a language system that gives
  - Understandable code
  - Performant code
  - Performance portability
- OpenCL does not really give this to us by itself
- What can we do?
  - For understandable code:
    - automatic code generation from a higher-level language
  - For performance and performance portability
    - empirical autotuning
OrCL Code Generation

- OrCL is implemented as part of the Orio framework
- The input is
  - A restricted subset of C …
  - … with annotations
    - come from OpenCL parameters of interest
- The output is OpenCL code
  - Can also produce C or CUDA code
Transformation Parameters

- The OpenCL code generator has a number of transformation parameters we can change to optimize performance

    /*@ begin Loop(transform OpenCL(
        workGroups=WG,
        workItemsPerGroup=WI,
        sizeHint=SH,
        vecHint=VH,
        cacheBlocks=CB,
        unrollInner=UI,
        clFlags=CL,
        device=0,
        platform=0)
    ...
Transformation Parameters

- `workGroups`
- `workItemsPerGroup`
  - Multiplied together, it is the total number of threads
  - If the data size is larger than the number of threads, each thread processes multiple data elements
  - Tradeoff between maximum use of computational resources and memory resources
  - More work items per workgroup = more parallelism and less local memory per work item
Transformation Parameters

- **sizeHint**
  - Controls OpenCL function attributes that give the compiler information on the expected or required local work size

- **work_group_size_hint**
  - Allows the compiler to make optimizations that improve performance when the local work size is equal to the hinted size, but that might degrade performance otherwise

- **reqd_work_group_size**
  - Allows the compiler to make optimizations that would yield incorrect results for sizes other than the hinted size
Transformation Parameters

- vecHint
  - Controls OpenCL function attribute that gives the compiler information on the basic computation width
    - `vec_type_hint(float2)`
  - Can assist in auto-vectorization
  - Can allow compiler to more intelligently schedule work items to make full use of available vector units
Transformation Parameters

- **cacheBlocks**
  - Whether to copy data from slower global memory into faster local memory before carrying out other computations
  - Increase performance, especially with non-unit-stride memory access
  - Also increases the memory consumption of each work item, which can lower the number that can be run per workgroup
Transformation Parameters

- **unrollInner**
  - Number of times to unroll the innermost loop of the kernel
  - Causes each work item to do more computation
  - Can help expose opportunities for autovectorization

- **clFlags**
  - Flags to control the OpenCL compiler’s optimizer
    - enable fused multiply-add
    - ignore signedness of zeroes
    - assume all floating-point arguments are defined
    - assume all floating-point arguments are finite
Searching the Transformation Parameters

- Wrap the transformation specification in a tuning specification

```plaintext
/*@ begin PerfTuning(
  def performance_params {
    param WI[] = [32,64,128,256];
    param WG[] = [4,8,16,32,64,128];
    param CB[] = [True, False];
    param SH[] = [True, False];
    param UI[] = range(1,4);
    param VH[] = [0,2,4];
    param CL[] = ['','-cl-fast-relaxed-math'][;]
  }
  def build {
    arg build_command = 'gcc -O3 -lOpenCL';
  }
  def input_params {
    param N[] = [100000,1000000];
  }
) @*/
```
Creating/Running Performance Experiment

- TAU wraps the OpenCL runtime, replacing OpenCL runtime functions with functions that call the original function before collecting OpenCL profiling data.
Initial Evaluation

- BLAS kernels on different accelerator devices

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>matVec</td>
<td>$y = Ax$</td>
</tr>
<tr>
<td>vecAXPY</td>
<td>$y = \alpha x + y$</td>
</tr>
<tr>
<td>vecMAXPY</td>
<td>$y = y + \alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_n x_n$</td>
</tr>
<tr>
<td>vecScale</td>
<td>$w = \alpha w$</td>
</tr>
<tr>
<td>vecWAXPY</td>
<td>$w = y + \alpha x$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accelerator Device</th>
<th>Radeon 6970</th>
<th>Radeon 7970</th>
<th>GTX 480</th>
<th>Tesla C2075</th>
<th>Tesla K20C</th>
<th>Xeon Phi</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenCL Version</td>
<td>1.2</td>
<td>1.2</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.2</td>
</tr>
<tr>
<td>Max Compute Units</td>
<td>24</td>
<td>32</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>204</td>
</tr>
<tr>
<td>Max Work Items</td>
<td>(256,256,256)</td>
<td>(256,256,256)</td>
<td>(1024,1024,64)</td>
<td>(1024,1024,64)</td>
<td>(1024,1024,64)</td>
<td>(1024,1024,1024)</td>
</tr>
<tr>
<td>Max Workgroup Size</td>
<td>256</td>
<td>256</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>880 MHz</td>
<td>1000 MHz</td>
<td>1401 MHz</td>
<td>1147 MHz</td>
<td>705 MHz</td>
<td>2000 MHz</td>
</tr>
<tr>
<td>Cache Size</td>
<td>None</td>
<td>16 KB</td>
<td>24 KB</td>
<td>224 KB</td>
<td>208 KB</td>
<td>None</td>
</tr>
<tr>
<td>Global Memory Size</td>
<td>1024 MB</td>
<td>2048 MB</td>
<td>1535 MB</td>
<td>5375 MB</td>
<td>4800 MB</td>
<td>2835 MB</td>
</tr>
<tr>
<td>Constant Buffer Size</td>
<td>64 KB</td>
<td>64 KB</td>
<td>64 KB</td>
<td>64 KB</td>
<td>64 KB</td>
<td>128 KB</td>
</tr>
<tr>
<td>Local Memory Size</td>
<td>32 KB</td>
<td>32 KB</td>
<td>48 KB</td>
<td>48 KB</td>
<td>48 KB</td>
<td>32 KB</td>
</tr>
<tr>
<td>Preferred Workgroup Size Multiple</td>
<td>64</td>
<td>64</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>
BLAS Evaluation Results Across Accelerators

![Graph showing BLAS performance evaluations across accelerators like Radeon, GTX, Tesla, Xeon, and CUDA.]
Comparison vs. ViennaCL

- We use *ViennaCL* as a benchmark against which to compare our performance
- A C++ linear algebra library for accelerator devices using OpenCL

```cpp
// Allocate CPU vectors:
std::vector<ScalarType> std_vec1(1000000);
std::vector<ScalarType> std_vec2(1000000);

// Allocate GPU vectors (and a scalar):
viennacl::vector<ScalarType> vcl_vec1(1000000);
viennacl::vector<ScalarType> vcl_vec2(1000000);
viennacl::scalar<ScalarType> vcl_s1 = ScalarType(5.0);

// Fill CPU vectors with random values:
for (unsigned int i = 0; i < 1000000; ++i)
{
    std_vec1[i] = random<ScalarType>();
    std_vec2[i] = random<ScalarType>();
}

// Copy the CPU vectors to the GPU
viennacl::copy(std_vec1.begin(), std_vec1.end(), vcl_vec1.begin());
viennacl::copy(std_vec2.begin(), std_vec2.end(), vcl_vec2.begin());

// Run the computation
vcl_vec2 += vcl_s1 * vcl_vec1;

// Copy the GPU vectors to the CPU
viennacl::copy(vcl_vec2.begin(), vcl_vec2.end(), std_vec2.begin());
```
Comparison vs. ViennaCL

ViennaCL better

Comparison vs. ViennaCL

- vecAXPY
- vecMAXPY
- vecWAXPY
- vecScale
- matVec

ViennaCL better
A More Complicated Example

- Solid fuel ignition solver
  \[-\nabla^2 u - \lambda e^u = 0 \text{ in } [0, 1] \times [0, 1] \times [0, 1]\]
  \[u = 0 \text{ on the boundary}\]

- Our goal is to replace two functions which represent a large proportion of overall execution time and offload them to the GPU

- The implementations take too long to run to exhaustively enumerate the search space
  - Use a Nelder-Mead search
Solid Fuel Ignition

```
int m = M;
int n = N;
int p = P;
int nrows = m*n*p;
double hx = 1.0/(m-1);
double hy = 1.0/(n-1);
double hz = 1.0/(p-1);
double sc = hx*hy*hz*lambda;
double hxzdhy = hx/hz/hy;
double hyzdhx = hy/hz/hx;
double hxhydhz = hx*hy/hz;
```

**FormFunction**

```
for(i=0; i<=nrows-1; i++) {
    if (i<m*n || i=nrows-m*n || i%(m*n)<m || i%(m*n)>=m*n-m || i%m==0 || i%m==m-1) {
        F[i] = X[i];
    } else {
        F[i] = (2*X[i] - X[i-1] - X[i+1])*hxzdhy
            + (2*X[i] - X[i-m] - X[i+m])*hxzdhx
            + (2*X[i] - X[i-m*n] - X[i+m*n])*hxhydhz
            - sc*exp(X[i]);
    }
}
```

**FormJacobian**

```
for(i=0; i<=nrows-1; i++) {
    if (i<m*n || i=nrows-m*n || i%(m*n)<m || i%(m*n)>=m*n-m || i%m==0 || i%m==m-1) {
        dia[i] = 1.0;
    } else {
        dia[i] = -hxhydhz;
        dia[i+nrows] = -hxzdhy;
        dia[i+2*nrows] = -hxzdhx;
        dia[i+3*nrows] = 2.0*(hxzdhx+hxzdhy+hxhydhz) - sc*exp(x[i]);
        dia[i+4*nrows] = -hxzdhx;
        dia[i+5*nrows] = -hxzdhy;
        dia[i+6*nrows] = -hxhydhz;
    }
}
```
FormFunction (different sizes, platforms)

- **FF 64x64x64**:
  - Radeon 6970
  - Xeon Phi
  - GTX 480
  - Tesla K20c
  - Radeon 7970
  - Tesla C2075

- **FF 75x75x75**:
  - Radeon 7970
  - Xeon Phi
  - GTX 480
  - Tesla K20c
  - Tesla C2075

- **FF 100x100x100**: 
  - Radeon 7970
  - Xeon Phi
  - Tesla K20c
  - Tesla C2075

- **FF 128x128x128**:
  - Radeon 7970
  - Xeon Phi
  - GTX 480
  - Tesla K20c
  - Tesla C2075
FormJacobian (different sizes, platforms)
**Factor Analysis**

- Parameter values for the best performing ex14FF and ex14FJ kernels across the architectures
  - Workgroups, Workitemspergroup, Unrollinner, Compilerflags, Sizehing, Vechint

<table>
<thead>
<tr>
<th>Accelerator Device</th>
<th>Radeon 6970</th>
<th>Radeon 7970</th>
<th>Xeon Phi</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex14FF 64³</td>
<td>(64,64,1,0)</td>
<td>(64,32,2,0)</td>
<td>(64,64,2,0)</td>
</tr>
<tr>
<td>ex14FF 75³</td>
<td>N/A</td>
<td>(16,32,4,0)</td>
<td>(32,128,2,0)</td>
</tr>
<tr>
<td>ex14FF 100³</td>
<td>N/A</td>
<td>(32,32,4,0)</td>
<td>(128,64,1,0)</td>
</tr>
<tr>
<td>ex14FF 128³</td>
<td>N/A</td>
<td>(32,64,4,0)</td>
<td>(16,256,1,0)</td>
</tr>
<tr>
<td>ex14FJ 64³</td>
<td>N/A</td>
<td>(64,64,2,0)</td>
<td>(128,64,2,0)</td>
</tr>
<tr>
<td>ex14FJ 75³</td>
<td>N/A</td>
<td>(64,256,1,0)</td>
<td>(64,256,2,0)</td>
</tr>
<tr>
<td>ex14FJ 100³</td>
<td>N/A</td>
<td>(128,128,2,0)</td>
<td>(16,32,2,0)</td>
</tr>
<tr>
<td>ex14FJ 128³</td>
<td>N/A</td>
<td>(32,128,2,0)</td>
<td>(32,64,4,0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accelerator Device</th>
<th>GTX 480</th>
<th>Tesla C2075</th>
<th>Tesla K20c</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex14FF 64³</td>
<td>(16,64,1,0)</td>
<td>(64,64,2,0)</td>
<td>(64,128,1,0)</td>
</tr>
<tr>
<td>ex14FF 75³</td>
<td>(128,64,4,0)</td>
<td>(16,128,2,0)</td>
<td>(32,32,2,0)</td>
</tr>
<tr>
<td>ex14FF 100³</td>
<td>(128,128,4,0)</td>
<td>(64,128,1,0)</td>
<td>(64,64,4,0)</td>
</tr>
<tr>
<td>ex14FF 128³</td>
<td>(32,32,2,0)</td>
<td>(64,128,2,0)</td>
<td>(32,64,1,0)</td>
</tr>
<tr>
<td>ex14FJ 64³</td>
<td>(16,64,2,0)</td>
<td>(64,128,1,0)</td>
<td>(32,128,1,0)</td>
</tr>
<tr>
<td>ex14FJ 75³</td>
<td>(64,256,2,0)</td>
<td>(128,64,2,0)</td>
<td>(32,256,1,0)</td>
</tr>
<tr>
<td>ex14FJ 100³</td>
<td>(32,128,2,0)</td>
<td>(32,64,1,0)</td>
<td>(32,64,4,0)</td>
</tr>
<tr>
<td>ex14FJ 128³</td>
<td>(32,128,4,0)</td>
<td>(32,64,2,0)</td>
<td>(32,64,4,0)</td>
</tr>
</tbody>
</table>
### Factor Analysis

<table>
<thead>
<tr>
<th>Workgroups</th>
<th>Xeon Phi</th>
<th>GTX 480</th>
<th>Tesla G2075</th>
<th>Tesla K20c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work Items</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler Flags</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unroll Factor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size Hint</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Hint</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lecture 9 – Parallel Performance Tools  
CIS 631: Advanced Parallel Computing
Hardware Performance Counters

- We would like to be able to attribute the performance of a variant to its effects on the hardware.
- We can do this for the Intel Xeon Phi for cache hits.

Here we see that the execution time is well-correlated with cache miss rate.
Hardware Performance Counters

- Getting detailed results is problematic
  - NVIDIA does not provide access to hardware performance counters for OpenCL code
  - AMD does expose some hardware performance counters, but only a very limited set
  - There is no standardized way to access hardware performance counters
Our Motivation Criteria

- Portability
  - Still yes, from OpenCL

- Understandable code
  - Still yes, from OrCL’s automatic code generation

- Performant code
  - Yes (generally better than ViennaCL)

- Performance portability
  - Yes for the more complicated FF and FJ kernels.
  - Somewhat for the simpler kernels (but better than ViennaCL).
Limitations and Future Work

- Hardware performance counters not portable
  - We plan to address this by instrumenting the output of the OpenCL compilers

- Limited number of parameterized transformations.
  - We plan to add more transformations.

- Lack of feedback of information from completed autotuning runs into future runs, and of performance models into the search.
  - We plan to develop an integrated knowledge management system to collect and make use of that type of information.
Related Work

- Concept of integrated measurement and analysis can be found in several parallel performance tools
  - Environments generally are not extensible
  - Do not generally provide interfaces for autotuning

- Several projects integrate data mining and machine learning techniques with compiler optimizations
  - Based on static properties, hardware counters, ...
  - Variety of classifiers

- Integration of machine learning and database to select optimizations across compilers and platforms
  - Collective Tuning (http://ctuning.org)
  - Collective Mind

- Intelligent compiler for predicting optimization
Collective Optimization Framework

Grigori Fursin, Inria

Synthesis for Automated Performance Tuning

- Integration of performance measurement and analysis with autotuning frameworks important

- An opportunity for synthesis of 2 types
  - Incorporate broader information from across the HPC development, performance, optimization, systems tools and technologies in the autotuning process
  - Preserve performance knowledge and enables higher-order understanding (learning) of the relationship between performance factors across tuning landscape

- SYNAPT is a concept to create a unified architecture for information synthesis and performance knowledge preservation