Towards Automating Black Belt Programming

Franz Franchetti
ECE, Carnegie Mellon University
www.ece.cmu.edu/~franzf

Co-Founder, SpiralGen
www.spiralgen.com

This work was supported by ONR, NSF, and ITRI
Giving Moore’s Law a Face

1 flop/s = one floating-point operation (addition or multiplication) per second

mega (M) = 10^6, giga (G) = 10^9, tera (T) = 10^{12}, peta (P) = 10^{15}, exa (E) = 10^{18}

In 2010...

Cell phone
1 Gflop/s

Laptop
20 Gflop/s

Workstation
1 Tflop/s

HPC
20 Tflop/s

#1 supercomputer
2.3 Pflop/s

...would have been the #1 supercomputer back in...

Cray X-MP/48
941 Mflop/s
1984

NEC SX-3/44R
23.2 Gflop/s
1990

Intel ASCI Red
1.338 Tflop/s
1997

Earth Simulator
35.86 Tflop/s
2002
How Big are the Computational Problems?

1 flop/s = one floating-point operation (addition or multiplication) per second
mega (M) = $10^6$, giga (G) = $10^9$, tera (T) = $10^{12}$, peta (P) = $10^{15}$, exa (E) = $10^{18}$

Matrix-matrix multiplication...

```
for i=1:n
  for j=1:n
    for k=1:n
      C[i,j] += A[i,k]*B[k,j]
```

..running on...

<table>
<thead>
<tr>
<th>System</th>
<th>FLOPS/s</th>
<th>Matrix Size</th>
<th>Memory</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell phone</td>
<td>1 Gflop</td>
<td>1k x 1k</td>
<td>8MB, 2s</td>
<td>2s</td>
</tr>
<tr>
<td>Laptop</td>
<td>20 Gflop</td>
<td>8k x 8k</td>
<td>0.5 GB, 5.5s</td>
<td></td>
</tr>
<tr>
<td>Workstation</td>
<td>1 Tflop</td>
<td>16k x 16k</td>
<td>2 GB, 8s</td>
<td></td>
</tr>
<tr>
<td>HPC</td>
<td>20 Tflop</td>
<td>64k x 64k</td>
<td>32 GB, 28s</td>
<td></td>
</tr>
<tr>
<td>#1 supercomputer</td>
<td>2.3 Pflop</td>
<td>1M x 1M</td>
<td>8 TB, 1,000s</td>
<td></td>
</tr>
</tbody>
</table>
Today’s Moving Target: Multicore CPUs

**IBM Cell BE**
8+1 cores

**Intel Core i7**
8 cores, 2-way SMT

**UltarSPARC T3**
16 cores, 8-way SMT

**IBM POWER7**
8 cores, 4-way SMT

**Intel SCC**
48 cores

**Nvidia Fermi**
448 cores, SMT

**Intel MIC**
50? cores
Outline

- Autotuning and program generation
- Black belt optimization of image segmentation
- Autotuning of image segmentation
- Performance evaluation
- Summary
Outline

- Autotuning and program generation
- Black belt optimization of image segmentation
- Autotuning of image segmentation
- Performance evaluation
- Summary
Programmer Demographics

**Black Belt Programmer**
- Performance libraries, device drivers
- C, Assembly, low-level tricks
- *Goal: performance above all*

**Software engineers**
- Large applications
- C++, Java, C#
- *Goal: Maintenance vs. performance*

**Average programmers**
- Small ad-hoc applications
- C, C++, Java, Perl, Python
- *Goal: implement their algorithm quickly*
Software’s Slow Pace of Change

Popular performance programming languages
- 1953: Fortran
- 1973: C
- 1985: C++
- 1997: OpenMP
- 2007: CUDA

Popular performance libraries
- 1979: BLAS
- 1992: LAPACK
- 1994: MPI
- 1995: ScaLAPACK
- 1995: PETSc
- 1997: FFTW

Popular productivity/scripting languages
- 1987: Perl
- 1989: Python
- 1993: Ruby
- 1995: Java
- 2000: C#
The Cost Of Portability and Maintainability

Matrix-Matrix Multiplication
Performance [Gflop/s]

Best GPU code
10,000 lines CUDA

6,500x =15 years technology loss

Best CPU code
100,000 lines, SSE, OpenMP

simple C code
4 lines
Compilers: The Fundamental Problem

Matrix-matrix multiplication

\[
\begin{align*}
&\text{for } i=1:N \\
&\quad \text{for } j=1:N \\
&\quad \quad \text{for } k=1:N \\
&\quad \quad \quad C[i,j] += A[i,k]\times B[k,j]
\end{align*}
\]

Tiled matrix-matrix multiplication

\[
\begin{align*}
&\text{for } i=1:NB:N \\
&\quad \text{for } j=1:NB:N \\
&\quad \quad \text{for } k=1:NB:N \\
&\quad \quad \quad \text{for } i0=i:NU:i+NB \\
&\quad \quad \quad \quad \text{for } j0=j:NU:j+NB \\
&\quad \quad \quad \quad \quad \text{for } k0=k:NU:k+NB \\
&\quad \quad \quad \quad \quad \quad \text{for } k00=k0:1:k0+NU \\
&\quad \quad \quad \quad \quad \quad \quad \text{for } j00=j0:1:j0+NU \\
&\quad \quad \quad \quad \quad \quad \quad \quad \text{for } i00=i0:1:i0+NU \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad C[i00,j00] += A[i00,k00]\times B[k00,j00]
\end{align*}
\]

Problem: which transformation order? What parameter values?
How to Overcome Compiler Limitations?

- **Autotuning**
  Write parameterized program and empirically find good parameters

  ```
  for (i = 0; i < 4; i++)
  A[i] = B[i] * C[i];
  ``

  Parameter: tile size, levels

- **Program generation**
  Automatically generate big basic blocks and special case code

  Parameter: which loop, unroll factor
Autotuning: From Simple to “Really Hard”

- **Level 0: simple C program**
  implements the algorithm cleanly

- **Level 1: C macros plus search script**
  use C preprocessor for meta-programming

- **Level 2: scripting for code specialization**
  text-based program generation, e.g., ATLAS

- **Level 3: add compiler technology**
  internal code representation, e.g., FFTW’s genfft

- **Level 4: synthesize the program from scratch**
  high level representation, e.g., TCE and Spiral
Two Kinds of Autotuning

- **Autotuning for black belt programmers**
  - Autotuning libraries
    - FFTW, ATLAS, Spiral, OSKI,…
  - Superoptimizer
  - Code generation scripts

- **Autotuning for the masses**
  - Many things can be autotuned
  - Autotuning compilers and frameworks
  - Important for lower-efficiency code

Proceedings of the IEEE special issue, Feb. 2005
Autotuning for Black Belt Programmers

- Understand algorithmic trade-offs
  - Cheaper iterations vs. fewer iterations
  - Computation load vs. algorithm regularity

- Understand the target hardware
  - What are the costs and trade-offs?
  - What are good implementation choices and tricks?

- Develop optimized parameterized implementation
  - Parameters should expose machine/algorithm interaction
  - Parameterization may require macros or program generation

- Autotune the algorithm and implementation parameters
  - Parameter choice is unclear because of algorithm/architecture interaction
  - Autotuning is the last step in aggressive performance tuning

100x gain: Optimized parameterized program: 50x, autotuning: 2x
Towards Automatic Black Belt Programming

- **Goal:** Empower software engineers
  Give software engineers tools that reach black belt performance and keep code maintainable

- **Study algorithm/hardware interaction**
  Dependencies are complicated and cannot easily be generalized

- **Build autotuning program synthesizers**
  Generalize proven approaches

**Big question:** how can we take black belt autotuning beyond signal processing and dense linear algebra?

*The (sad) answer: no magic, only one step at a time*
Outline

- Autotuning and program generation
- Black belt optimization of image segmentation
- Autotuning of image segmentation
- Performance evaluation
- Summary

PhD thesis of Wei Yu, co-advised with James C. Hoe
Level-Set Based Image Segmentation

**Level Set Algorithm**
- PDE-based image segmentation method
- Image produces force field on moving contour
- Embeds contour into n+1 dimensional level-set function

**Narrow Band Level Set Algorithm**
- Compute only in neighborhood of boundary
- Makes algorithm computationally tractable but complicated
Target: Cache-Based Multicores

COTS Multicore

- Shared memory (SMP or NUMA)
- Cache-based memory (L1, L2, L3)
- SIMD vector instructions (SSE, AltiVec, ...)

IBM POWER7
8 cores, 4-way SMT

Intel Core i7
8 cores, 2-way SMT

UltraSPARC T3
16 cores, 8-way SMT

Intel Atom
1 core, 2-way SMT
Why A Challenging Problem?

SparseMV

Narrow Band Level Set

Stencil

Memory bound

\[ O(1) \]

compute bound

\[ O(N) \]

sparse & irregular data structure

sparse data with dense substructure

dense & regular data structure

compute bound

\[ O(N) \]

sparse & irregular data structure

sparse data with dense substructure

dense & regular data structure
How to Get Level Set Up To Speed?

**Sparse Linear Algebra**

\[
\begin{bmatrix}
A
\end{bmatrix}
\times
\begin{bmatrix}
x
\end{bmatrix}
=
\begin{bmatrix}
y
\end{bmatrix}
\]

**Code generation**

```c
for (int j=lj; j<=hj; j++)
    for (int i=li; i<=hi; i++)
        BI[j][i]=1;
```

```c
switch (case) {
    case (...):
        BI[j-1][i-1]=1;
        ...
        BI[j+1][i+1]=1;
        break;
    case (...):
        ...
}
```

**Time Skewing**

**Auto-tuning**

![Parameter 1 vs. Parameter 2](Parameter 1 vs. Parameter 2)

![Parameter 3](Parameter 3)
Narrow Band Level Set in a Nutshell

\[ \phi^{t+1} = \Gamma(\phi^t (x \pm \Delta_x, y \pm \Delta_y)) \]

```
<table>
<thead>
<tr>
<th>Zero level set at t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero level set at t+1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Narrow band</th>
</tr>
</thead>
<tbody>
<tr>
<td>Narrow band</td>
</tr>
</tbody>
</table>
```

Autotuning parameter: narrow band width
Projection of Manifold into Dense Stencil

Autotuning parameter: tile size
Projective Time Skewing

Standard 1-D dense time-skewing

Polyhedral width

Height

Time iteration

Projective time-skewing

2D row number

$t_0$

$r_0$ $r_1$ $r_2$ $r_3$ $r_4$ $r_5$ $r_6$

$t_0+1$

$r_0$ $r_1$ $r_2$ $r_3$ $r_4$ $r_5$ $r_6$

$t_0+2$

$r_0$ $r_1$ $r_2$ $r_3$ $r_4$ $r_5$ $r_6$

$t_0+3$

$r_0$ $r_1$ $r_2$ $r_3$ $r_4$ $r_5$ $r_6$

Autotuning parameter: polyhedral tile size
In-Core Stencil Optimization

Optimizations
- Common subexpression elimination
- Transcendental function approximation (cos by polynomial)
- SIMD vectorization
- Instruction scheduling (intra and inter stencil)

Autotuning parameter: tile instruction schedule
Multicore Optimization: Software Pipeline

Prologue

Core 1

Core 2

Steady state

Epilogue

Data transfer between cores requires a memory fence

Autotuning parameter: prologue/epilogue vs. steady state
Narrow Band Update: Program Generation

```c
// rubber stamp zero level set
for (i=0;i<TILE;i++)
    for (j=0;j<TILE;j++)
        if (!is_zero(Ti+i,Tj+j)) {
            for (k=-BAND;k<=BAND;k++)
                for (m=-BAND;m<=BAND;m++)
                    band[Ti+i+k][Tj+j+m]=1;
        }
```

// TILE=4x4, BAND=+/-1
// unroll: 2x4
for (i=0;i<4;i+=2) {
    b0=&band[Ti+i-1][Tj-1];
    b1=&band[Ti+i][Tj-1];
    b2=&band[Ti+i+1][Tj-1];
    b3=&band[Ti+i+2][Tj-1];
    switch (zeropat(Ti+i,Tj)) {
        ...
        case PAT(0,1,1,0,0,0,1,0):
            b0[1]=1;b0[2]=1;b0[3]=1;b0[4]=1;
            break;
        case PAT(...):
    }
```

Autotuning parameter: rubber stamp unrolling
Outline

- Autotuning and program generation
- Black belt optimization of image segmentation
- Autotuning of image segmentation
- Performance evaluation
- Summary
A Fundamental Tradeoff

runtime

Narrow band radius
## Parameter Space

<table>
<thead>
<tr>
<th>Tuning Parameters</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tile size</td>
<td>{1,2,4,8}x{4,8,16}</td>
</tr>
<tr>
<td>Band radius</td>
<td>{1,2,3,4,5,6,7,8}</td>
</tr>
<tr>
<td>Polytope size height</td>
<td>{2, 4, 8, 16, 24, 32, 48, 64, 80, 96, 128, 160, 192}</td>
</tr>
<tr>
<td>Polytope size height</td>
<td>{2, 4, 8, 12, 16, 32}</td>
</tr>
<tr>
<td>Enable/disable</td>
<td>SIMD, instruction scheduling, approx. <code>cos()</code>, padding, large pages,...</td>
</tr>
</tbody>
</table>
Autotuning: Iterative Line Search

Iterative Line search
- Search parameters one by one hold others fixed
- Multiple refinement iterations
- Search space is well-behaved

Heuristics
- Cut search space
- For instance: DAG scheduling
Outline

- Autotuning and program generation
- Black belt optimization of image segmentation
- Autotuning of image segmentation
- Performance evaluation
- Summary
Speedup on 2.8 GHz Dual Quad-Core Xeon

speed-up

image size

128 256 512 1,024 2,048 4,096 8,192

+threading
+program generation
+large pages
+projective time tiling
+DAG scheduling
+SIMD
+cos approx, rsqrt
+narrow band tuning
Computational Efficiency

Performance [Gflop/s]

Peak performance (single core)

In-core kernel upper bound

Full program

C base line

Image size
Parallel Speed-Up On Xeon

- **8 threads**
- **4 threads**
- **2 threads**

**speed-up**

- 8
- 7
- 6
- 5
- 4
- 3
- 2
- 1

**image size**

- 128
- 256
- 512
- 1,024
- 2,048
- 4,096
- 8,192
Performance Gain Through Autotuning

speed-up

image size

Atom

Xeon

64 128 256 512 1,024 2,048 4,096 8,192
Outline

- Autotuning and program generation
- Black belt optimization of image segmentation
- Autotuning of image segmentation
- Performance evaluation
- Summary
Lessons For Black Belt Programming

- 10x to 100x speed-up is often possible
  really hard work, specific to the problem

- Doable with standard tools
  C compiler, autotuning, and program generation scripts

- Crucial: extract meaningful parameters
  capture machine/algorithm interaction, one algorithm a time

- Many things must be done right
  SIMD, threading, low-level tricks, specialization, tuning

- Autotuning gives the last 50% to 2x speed-up
  the remaining 5x to 50x come from standard optimization
Onwards, Autotuners!

http://www.ece.cmu.edu/~franzf