Empirical Auto-tuning Code Generator for FFT and Trigonometric Transforms

Ayaz Ali and Prof. Lennart Johnsson
Texas Learning and Computation Center
University of Houston

Dr. Dragan Mirkovic
MD Anderson Cancer Center
University of Texas, Houston
Outline

- Motivation
- Introduction
- Empirical Optimization
- Results
- Conclusion
End of Moore’s Law has pushed the trend towards Specialized architectures.

Result:
- Increasingly difficult for compilers to perform optimizations.
- Compilers have always lagged the Hardware Technology Advancements.
Hardware/Software Gap

- Traditional ways to Achieve Higher Performance
  - Vendor compilers.
  - Source to source transformations, mainly targeting loops.
  - Hand Tuning.

- Compilers are not good at optimizing even the regular Matrix Multiplication (MMM).
Adaptive Auto-tuning Libraries

- **Auto-tuning** performed in two stages.
  - **Code Generator** adapts to the *Microprocessor Architecture* by generating highly optimized blocks of codes (micro-kernels) at installation time.
  - **Run-time** adapts to the *Memory System* by scheduling the execution of those blocks.

- **Examples**: ATLAS, FFTW, UHFFT etc.
Adaptive FFT Libraries

- **UHFFT and FFTW**
  - Two Stage Optimization
    - The Code Generator
      “generates microkernels of optimized straight-line C code blocks (radix/codelets) that compute part of the FFT problem”
    - The Run-Time
      “schedules the execution of a FFT problem by composing parameterized codelets generated by code generator at the installation time”

- **SPIRAL**
  - Single Stage (Code generator system).
Tuning FFT

- Many formulas to represent a single FFT (Different Operation Counts).
- FFT’s Cache oblivious schedule allows register blocking but it can be improved for larger size codelets.
- Because of the recursive schedule, loop level optimizations are not as effective as in BLAS.
- Strided (Non-sequential) data access in FFT.
FFT Formula

Many formulas to represent FFT (e.g. size 12).

![Diagram showing FFT Formula's impact on performance. Lower OpCount does not always lead to higher performance.](image)
Instruction Scheduling

### Two versions of Codelet size 4. Upto 7% Improvement.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Compiler</th>
<th>MFLOPS (Ver:1)</th>
<th>MFLOPS (Ver:2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Itanium2(1.5GHz)</td>
<td>gcc</td>
<td>1815</td>
<td>1878 (3-4%)</td>
</tr>
<tr>
<td>Itanium2</td>
<td>icc 8</td>
<td>1932</td>
<td>2065 (7%)</td>
</tr>
<tr>
<td>Itanium2</td>
<td>icc 9</td>
<td>1932</td>
<td>1997 (3-4%)</td>
</tr>
<tr>
<td>Opteron(2.0GHz)</td>
<td>Pathscale</td>
<td>2449</td>
<td>2597 (5-6%)</td>
</tr>
<tr>
<td>Opteron</td>
<td>gcc</td>
<td>2372</td>
<td>2500 (5-6%)</td>
</tr>
</tbody>
</table>
# Code Generation (UHFFT1.6 vs FFTW3)

<table>
<thead>
<tr>
<th></th>
<th>FFTW</th>
<th>UHFFT1.6 (older)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT Formula Minimization</td>
<td>Heuristics based</td>
<td></td>
</tr>
<tr>
<td>Schedule</td>
<td>Cache Oblivious followed by DAG reordering</td>
<td>Cache Oblivious schedule of instructions</td>
</tr>
<tr>
<td>Register Blocking</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Architecture (ISA)</td>
<td>Yes (e.g. fma, sse etc.)</td>
<td>Relies on compiler completely.</td>
</tr>
<tr>
<td>Specific Codelets</td>
<td>Straight-line and single loop (vector-recursion)</td>
<td>Straight-line</td>
</tr>
<tr>
<td>Codelet Types</td>
<td>Written Caml (rule based language)</td>
<td>Written in C (allows more flexibility)</td>
</tr>
<tr>
<td>Generator</td>
<td>Expert User selects the best after some trials</td>
<td>Expert User selects the best after some trials</td>
</tr>
</tbody>
</table>
Empirical Optimization

- Iterative Empirical Optimization through Compiler Feedback.
  - Auto-tune Non-deterministic elements in performance of codes.
  - Adapt to Architecture as well as compiler.
  - Recently: research focusing Iterative compiler optimization of whole applications.
  - Successfully used in ATLAS for BLAS codes.
  - Costly but manageable (esp. in microkernels).
Empirical Optimization

- Basic Ingredient (Iterative Compilation Loop)
  - Generate Variants (varying parameters)
  - Dynamic Compilation
  - Evaluate performance (runtime)
  - Guide tuning process
UHFFT Empirical Auto-tuning Code Generator
**FFDL Grammar**

<table>
<thead>
<tr>
<th>#</th>
<th>Productions</th>
</tr>
</thead>
</table>
| 1-4 | FFT $\rightarrow$ Module  
|     |   | FFTMR  
|     |   | FFTSR  
|     |   | FFTPF |
| 5-6 | FFTPF $\rightarrow$ FFTPF $p_{fa}$ Module  
|     |   | Module |
| 7-9 | Module $\rightarrow$ [rader, FFT] $z$  
|     |   | [FFT] $z$  
|     |   | Codelet |
| 10  | FFTMR $\rightarrow$ FFT $m_{r}$ Module |
| 11  | FFTSR $\rightarrow$ [sr Module, Module] $z$ |
| 12  | Codelet $\rightarrow$ {2,3,4,…,16,32,64,…} |
| 13  | $z$ $\rightarrow$ Z |

**GRAMMAR**
FFT Formula Description Language (FFDL) to specify the FFT Formula.
Codelet Types

Compact String representation of Codelet Types
Codelet Types

- Twiddle Codelet
  - Perform FFT and multiply with twiddle factors.

- Vector Codelets
  - Compute multiple FFTs on 2D blocks by pushing loop inside the codelets.

- Rotated Codelets
  - For PFA algorithm (Less Multiplications; No Twiddle Multiplications)

```plaintext
CodeletN(...)
{
    For (i=1 to r)
        compute fftN
}
```
Level 1 Optimizations

<table>
<thead>
<tr>
<th>Variants</th>
<th>Left Recursion</th>
<th>Rader Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-3</td>
<td>0</td>
<td>CIRC, SKEW, SKEWP</td>
</tr>
<tr>
<td>4-6</td>
<td>1</td>
<td>CIRC, SKEW, SKEWP</td>
</tr>
</tbody>
</table>

Left Recursive Factorization

Right Recursive Factorization

<table>
<thead>
<tr>
<th>Mode</th>
<th>Rader Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIRC</td>
<td>Circulant Permutation</td>
</tr>
<tr>
<td>SKEW</td>
<td>Skew Circulant</td>
</tr>
<tr>
<td>SKEWP</td>
<td>Skew Circulant with Partial Diagonalization</td>
</tr>
</tbody>
</table>
Level 1 Optimizations

Performance Variation (Level 1)
Itanium2 gcc -O2

"MFLOPS" vs Codelets

Real FFT
Level 2 Optimizations

<table>
<thead>
<tr>
<th>Variants</th>
<th>Reverse</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-3</td>
<td>0</td>
<td>2,4,8</td>
</tr>
<tr>
<td>4-6</td>
<td>1</td>
<td>2,4,8</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{Re}(y[os]) &= \text{Re}(x[0]) - \text{Re}(x[is]) \\
\text{Im}(y[0]) &= \text{Im}(x[0]) + \text{Im}(x[is]) \\
\text{Re}(y[0]) &= \text{Re}(x[0]) + \text{Re}(x[is]) \\
\text{Im}(y[os]) &= \text{Im}(x[0]) - \text{Im}(x[is]) \\
\end{align*}
\]
Level 2 Optimizations

Performance Variation (Level 2)
Itanium2 gcc -O2

"MFLOPS"

Real FFT
Level 3 Optimizations

\[ Y[0] = X[0] + X[1]; \]
\[ Y[1] = X[0] - X[1]; \]

\[ \downarrow \]

\[ \text{reg0} = X[0]; \]
\[ \text{reg1} = X[1]; \]
\[ \text{reg2} = \text{reg0} + \text{reg1}; \]
\[ \text{reg3} = \text{reg0} - \text{reg1}; \]
\[ Y[0] = \text{reg2}; \]
\[ Y[1] = \text{reg3}; \]
Level 3 Optimizations

Performance Variation (Level 3)
Itanium2 gcc -O2

Real FFT
Results

- Experimental Setup:
  - Two Architectures (Itanium2 and Opteron)
  - Two Compilers on each Architecture:
    - icc/gcc and pathcc/gcc

- Performance Metric:
  - “MFlop/s”
  - Vary Codelet strides to mimic actual execution inside the FFT schedule. (Larger strides => Lower Performance).
Codelets (Itanium)
Codelets (Opteron)
Codelet Size vs Performance

Itanium has more registers

Opteron has larger instruction cache
Performance Improvement (UHFFT2)

- FFT (Powers of 2 Size) on Itanium2 using ICC

Major Gain due to Vector Codelets
Performance Improvement (UHFFT2)

- FFT (Non-Powers of 2) on Itanium2 using ICC

**Chart:**
- **Performance (Non Powers 2)**
- Itanium2/1.5GHz

**Graph Notes:**
- Major Gain due to Special PFA Codelets
Performance Improvement (UHFFT2)

- FFT (Powers of 2 Size) on Itanium2 using GCC

![Graph showing performance improvement with major gain due to scalar replacement](image-url)
Performance Improvement (UHFFT2)

- FFT (Non-Powers of 2) on Itanium2 using GCC

Major Gain due to Special PFA Codelets
Conclusion

- Mixed Results; this technique generates codelets at least as good as any heuristics.
- Better the Compiler => lesser the impact.
- Performance portability across compilers.
- Major performance improvement for larger size codelets but large codelets hardly used because of lower performance.
- Simple heuristics could be used in some cases instead of expensive empirical auto-tuning specially for powers of two codelets.
Thank You!
Baseline Performance

- UHFFT Performance (Prior to Empirical Tuning)
Baseline Performance

Special Codelets (e.g. fma etc) in FFTW and MKL

UHFFT slightly better at managing memory
## Code Generation
*(FFTW3 vs UHFFT2)*

<table>
<thead>
<tr>
<th></th>
<th>FFTW</th>
<th>UHFFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT Formula Minimization</td>
<td>Uses Heuristics</td>
<td>Try limited number of (heuristics based) formulas plus any user supplied formulas</td>
</tr>
<tr>
<td>Schedule</td>
<td>Cache Oblivious followed by DAG reordering</td>
<td>Cache Oblivious schedule followed by DAG reordering</td>
</tr>
<tr>
<td>Register Blocking</td>
<td>Yes</td>
<td>Multiple blockings empirically evaluated</td>
</tr>
<tr>
<td>Architecture (ISA) Specific Codelets</td>
<td>Yes (e.g. fma, sse etc.)</td>
<td>No (Relies on compiler completely).</td>
</tr>
<tr>
<td>Codelet Types</td>
<td>Straight-line and looped FFT, Twiddle and In-place (transposed) codelets (No PFA)</td>
<td>Straight-line and looped FFT, Twiddle, Inplace and rotated PFA codelets.</td>
</tr>
<tr>
<td>Generator</td>
<td>Written Caml (rule based language)</td>
<td>Written in C (allows more flexibility)</td>
</tr>
<tr>
<td>Adaptation</td>
<td>Adapt to Architecture</td>
<td>Adapt to both Architecture and Compiler</td>
</tr>
<tr>
<td>Optimizations</td>
<td>Based on Heuristics (Trial and Error Method for more options)</td>
<td>Empirically Tuned Automatic Code generator</td>
</tr>
</tbody>
</table>
Our Solution

- Generate more and specialized microkernels.
- Consider more FFT Formulas.
- Use register blocking to avoid register pressure.
- Use Iterative Empirical Optimization through Compiler Feedback.