Scheduling FFT Computation on SMP and Multicore Systems

Ayaz Ali, Lennart Johnsson & Jaspal Subhlok

Texas Learning and Computation Center
Department of Computer Science
University of Houston
Outline

• Motivation
• Introduction
• FFT Background
• Parallelization Challenges
• Performance Results
• Conclusion
Motivation

• FFT is one of the most popular algorithms in scientific and engineering applications
  - More than 40% if we include all signal processing applications in modern telecommunications.

• Other Applications include:
  - Digital Image Processing and Compression.

• In all these applications, FFT takes a significant part of CPU time.

• Strong motivation for development of highly optimized implementations.
Motivation

• **Optimization Challenges:**
  - *Growing complexity of Modern Architectures:*
    - Deep memory hierarchy.
    - Multilevel Parallelism.
  - Compilers technology can not fill all the gaps.

• **Hand Tuning:**
  - Tedious (~200 lines of c code for size 8 FFT).
  - Expensive due to growing heterogeneity.

• **Solution:**
  - Autotuning through domain specific frameworks.
  - Automatic scheduling of operations.
Automatic Tuning

• **Automatic Performance Tuning Approach**
  - Two Stage Methodology (widely used)

• **Installation time**
  - Generate optimized code blocks (micro-kernels).
  - Optimized for instruction schedules and registers use.

• **Run time**
  - Select and schedule variants (algorithms, factors).
  - Optimized for memory access schedule and load balancing.

• **Examples:**
  - FFT: MKL, FFTW, UHFFT
  - Linear Algebra: ATLAS
In this paper...

- Extend the Auto-tuning approach in UHFFT to Multi-core and SMP architectures.
  - OpenMP and PThreads implementations
- Establish the heuristics for selecting the best schedule (plan) of computation and factorization.
  - Different factorizations have different memory access pattern.
- Empirical evaluation of parallel FFT on latest architectures.
  - SMP: Itanium2 and Opteron
  - CMP: Xeon and Opteron
A Brief Introduction to FFT
Mathematical Background

- The Fast Fourier Transform (FFT):
  - Faster algorithm for evaluation of the Discrete Fourier Transform (DFT).

- DFT is a matrix vector product $\Theta(N^2)$

$$y_j = \sum_{k=0}^{N-1} \omega_N^{jk} x_k$$

$$Y = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \cdot X$$

DFT Matrix size $N=4$
Mathematical Background

- **FFT algorithm**
  - Exploits the symmetries in the DFT matrix.
  - The famous Cooley Tukey Algorithm (1965) \( \Theta(N \log N) \)
  - Divide the Problem into smaller non-unique factors:

\[
N = m \times r
\]

\[
F_N = (F_r \otimes I_m).T^m_n.(I_r \otimes F_m).P^n_r
\]

\[
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
y_3 \\
\end{bmatrix} = \begin{bmatrix} 1 & 1 \\
1 & 1 \\
1 & \omega_2^1 \\
1 & \omega_2^1 \\
\end{bmatrix} \begin{bmatrix} 1 & 1 \\
1 & \omega_4^0 \\
1 & \omega_4^1 \\
1 & \omega_4^1 \\
\end{bmatrix} \begin{bmatrix} 1 & 1 \\
1 & \omega_2^1 \\
1 & \omega_2^1 \\
1 & \omega_2^1 \\
\end{bmatrix} \begin{bmatrix} x_0 \\
x_2 \\
x_1 \\
x_3 \\
\end{bmatrix}
\]

**FFT N=4**

\( m=2 \) \( r=2 \)
Mathematical Background

Strided Access

\[
\begin{bmatrix}
  y_0 \\
  y_1 \\
  y_2 \\
  y_3
\end{bmatrix} =
\begin{bmatrix}
  1 & 1 \\
  1 & \omega_2^1 \\
  1 & \omega_2^1 \\
  1 & \omega_2^1
\end{bmatrix}
\begin{bmatrix}
  1 \\
  1 \\
  1 \\
  1
\end{bmatrix}
\begin{bmatrix}
  1 & 1 \\
  1 & \omega_2^1 \\
  \omega_4^0 & \omega_4^1 \\
  \omega_4^0 & \omega_4^1
\end{bmatrix}
\begin{bmatrix}
  x_0 \\
  x_2 \\
  x_1 \\
  x_3
\end{bmatrix}
\]

Twiddle Multiplication

Butterfly Visualization

Bit Reversal

Ayaz Ali

21st International Conference on Supercomputing, Seattle
FFT Challenges

- **Algorithmic**
  - Unfavorable data access pattern \(2^n\) strides).
  - Low floating point vs. loads & stores ratio (Bandwidth bound).
  - Unbalanced Multiplication/Addition operation count.

- **Compare that with Linear Algebra Codes**
  - Low Spatial Locality.
  - Low Algorithm Efficiency
  - Codelet size=16
  - High Spatial Locality.
  - High Algorithm Efficiency
  - Block size=16
Strided Data Access

• Performance Impact.

66% Drop for stride of only 1K
UHFFT Design

UHFFT Installation

Codelet Generator
- Butterfly Computation
  - Factorization
  - Optimization (Scalar and Vector)
  - Scheduler (DAG)
  - Unparser (Output)

Compile - Evaluate and Guide (AEOS)

UHFFT Run-Time

Input FFT Problem

UHFFT Run-Time
- DFTi API
- FSSL Grammar (Parser / Scanner)

Executor
- SMP Layer
  - Split Radix
  - Prime Factor
  - Rader’s
- Mixed Radix

Planer
- Choose Algorithm
- Select Codelet
- Evaluate Performance
- Timers

Best FFT Descriptor Plan
- Single CPU
- SMP

Butterfly Computation
- Algorithm Selection

21st International Conference on Supercomputing, Seattle
Parallelization of FFT on SMP/CMP
Summary

• **FFT**
  - Unfavorable strided memory access.
  - Bit reversal permutation required for in-order result.
  - Many factorizations (schedules) to solve a given FFT problem.
  - Divide and Conquer Algorithm Complexity $O(n \log n)$

• **Architectures**
  - Trend towards Deeper memory hierarchy
  - Growing gap between processing power and memory speed/bandwidth with multi-cores.
  - Multi-cores with shared or private cache.
Parallelization of FFT

- FFT is a naturally data parallel algorithm.
- Parallelize at the top level of recursive tree (2D formulation).
Parallelization of FFT

• Only one rendezvous point.

Example

N=16
m=4
r=4

Sync here
Barrier or Transpose
Choice of Factors

- Where to put the barrier in FFT Plan?
  - Between the row \((r)\) and column \((m)\) FFT stages.
  - \(FFT_m\) is performed recursively using smaller factors.
  - \(FFT_r \in \{\text{codelet library generated at installation time}\}\).
Choice of Factors

- **Work distribution**
  - Perfect load balancing possible by choosing the factors $m$ and $r$ that are divisible by number of Threads $P$. ($m|P$ and $r|P$)

- **Choose largest column blocking to resolve cache coherence issues.**

Itanium 2 (4way SMP)
FFT: N=64K Complex Double
Multithreading Overhead

- **Two Main Sources**
  - **Thread Synchronization**
    - Use Busy wait to bypass thread synchronization OS calls.
    - Load Balance
  - **Thread Creation**
    - Use Thread Pooling
Thread Pooling

• Create the worker threads at the start.
  - Master thread notifies when the task is ready.
• The other model is fork-join model.
  - Overhead of creating and destroying threads.
Thread Pooling Performance

Xeon Woodcrest (2xDual CMP/SMP)
FFT: $N=2^n$ Complex Double $P=4$ cores
Performance Results
## Platforms

<table>
<thead>
<tr>
<th></th>
<th>Itanium 2</th>
<th>Opteron 846</th>
<th>Xeon Woodcrest</th>
<th>Opteron 275</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration</td>
<td>4 Processor SMP</td>
<td>8 Processor SMP</td>
<td>2×Dual Core CMP/SMP</td>
<td>2×Dual Core CMP/SMP</td>
</tr>
<tr>
<td>CPU Speed</td>
<td>1.5 GHz</td>
<td>2.0 GHz</td>
<td>2.66 GHz</td>
<td>2.2 GHz</td>
</tr>
<tr>
<td>Data Cache</td>
<td>16K,256K,6M</td>
<td>64K,1M</td>
<td>32K/Core,4MB/Duo</td>
<td>64K/Core,1MB/Core</td>
</tr>
<tr>
<td>Line Size</td>
<td>64B,128B,128B</td>
<td>64B,64B</td>
<td>64B,64B</td>
<td>64B,64B</td>
</tr>
<tr>
<td>Associativity</td>
<td>4,8,12 way</td>
<td>2,16 way</td>
<td>8,16 way</td>
<td>2,16 way</td>
</tr>
<tr>
<td>Theoretical Peak</td>
<td>24 GFLOPS</td>
<td>32 GFLOPS</td>
<td>42.56 GFLOPS</td>
<td>17.6 GFLOPS</td>
</tr>
<tr>
<td>Compilers</td>
<td>icc9.1</td>
<td>pathcc2.5</td>
<td>icc9.1</td>
<td>gcc4.1</td>
</tr>
</tbody>
</table>

SMP

SMP/CMP
Multi-core Cache Configuration

Xeon Woodcrest (2xDual CMP/SMP)

Processor 0
- Core 0 CPU
  - L1 Cache
  - L2 Cache

Processor 1
- Core 1 CPU
  - L1 Cache
  - L2 Cache

Opteron 275 (2xDual CMP/SMP)

Processor 0
- Core 0 CPU
  - L1 Cache
  - L2 Cache

Processor 1
- Core 1 CPU
  - L1 Cache
  - L2 Cache
Shared/Private Cache

Xeon Woodcrest (2xDual CMP/SMP)  
FFT: N=2^n Complex Double P=4 cores

Opteron 275 (2xDual CMP/SMP)  
FFT: N=2^n Complex Double P=4 cores

In each graph problem was executed ten times and average was plotted with the variation bar. Two lines in each graph represent the number of threads spawned.

Ayaz Ali  
21st International Conference on Supercomputing, Seattle
Shared/Private Cache

- Although (Linux kernel 2.6) fixed some of the thread scheduling issues (e.g. ping pong).
- To ensure best performance:
  - Set the CPU affinity to thread specifically in a program `sched_setaffinity` or `KMP_AFFINITY` (intel)
Programming Models

• **OpenMP:**
  - Portable Multithreaded programming model.
  - Parallelization performed by compiler.
  - OpenMP Implementations optimized by vendors.

• **PThreads:**
  - Allows more flexibility.
  - Parallelization defined by user.

• For a carefully partitioned data both OpenMP and PThreads can achieve identical performance.
OpenMP vs PThreads

- Performance of one, two and four threads using OpenMP and PThreads implementations in UHFFT.
- Barrier implementation customized for PThreads using busy wait and atomic decrement.
UHFFT and FFTW

- Support for efficient CMP/SMP FFT
  - Recently: UHFFT2.0.1-beta and FFTW3.2-alpha
- Similar adaptive methodology (code generation and run-time search).
- Many Implementation level differences, mainly:
  - FFTW generates ISA specific codelets (i.e. sse, fma etc.). UHFFT use automatic empirical optimization code generator.
  - UHFFT better at cache blocking.
UHFFT vs FFTW

FFTW utilizes threading for \( N > 256 \).
Overall Efficiency

- Itanium and Opteron:
  - Increase in total cache proportional to increase in cores.

- Xeon:
  - Shared cache.
Conclusions

• The best sequential FFT plan is not guaranteed to perform optimally on multi-cores.
  – Factorization has to take into account cache coherence related issues and load balancing.
• Given a carefully scheduled FFT computation both OpenMP and PThreads can perform equally well.
• Performance benchmarks on the most recent architectures show good speedup using our implementation in UHFFT.
• But: For FFT, high efficiency can only be possible if the memory bandwidth is increased proportional to the processing power.
Thank You!
Backup Slides
Low Efficiency of FFT

Performance (Powers of 2)
Itanium2/1.5GHz

Size

MFLOPS

5.00E+03
4.00E+03
3.00E+03
2.00E+03
1.00E+03
0.00E+00

8 64 512 4096 32768 2048 2097152 16777216 1.34E+08
## Algorithms

**Op-Counts**
Effect of Algorithm Selection on Op-Count.

<table>
<thead>
<tr>
<th>Transform Size</th>
<th>Mixed Radix (MR)</th>
<th>MR+PFA</th>
<th>MR+PFA+SplitRadix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Adds</td>
<td>Mults</td>
<td>Adds</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>40</td>
<td>16</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>92</td>
<td>52</td>
<td>84</td>
</tr>
<tr>
<td>8</td>
<td>52</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>80</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>108</td>
<td>40</td>
<td>100</td>
</tr>
<tr>
<td>11</td>
<td>236</td>
<td>116</td>
<td>220</td>
</tr>
<tr>
<td>12</td>
<td>104</td>
<td>32</td>
<td>96</td>
</tr>
<tr>
<td>13</td>
<td>232</td>
<td>108</td>
<td>216</td>
</tr>
<tr>
<td>14</td>
<td>224</td>
<td>128</td>
<td>196</td>
</tr>
<tr>
<td>15</td>
<td>196</td>
<td>88</td>
<td>180</td>
</tr>
<tr>
<td>16</td>
<td>148</td>
<td>28</td>
<td>144</td>
</tr>
<tr>
<td>32</td>
<td>388</td>
<td>108</td>
<td>372</td>
</tr>
<tr>
<td>64</td>
<td>964</td>
<td>332</td>
<td>912</td>
</tr>
</tbody>
</table>
### Data Distribution (SMP/CMP)

**Example**

\[ N = 16 \]
\[ m = 4, \ r = 4 \]
\[ P = 2 \]

#### Row FFTs

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

#### Column FFTs

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
UHFFT2 Current Status

- 1D Complex
- Out-of-place/In-place in-order
- Forward scrambled (Out-of-place/In-place)
- Forward/Inverse (Configurable Sign)
- High, Medium and Low Effort Plan Search
- Single/Double Precision
- SMP/CMP Executor (DFTi Extension)
- Empirical Auto-tuning code generator.
- Executor Extendibility through FSSL Grammar.
- Real FFT (not-integrated)
Future Work

• Executor:
  – Multidimensional
  – MPI

• Planner:
  – Accurate Model Driven Plan Search Scheme.
  – Economical (time and memory).
  – Schedule memory transactions through pre-fetching.

• Code Generator:
  – ISA specific codelet generation (e.g. sse, fma etc.).
  – Generate codelet cost models.