How to Write Fast Numerical Code
Spring 2016
Lecture: Autotuning and Machine Learning

Instructor: Markus Püschel
TA: Gagandeep Singh, Daniele Spampinato, Alen Stojanov

Overview

- Rough classification of autotuning efforts seen in course
- Use of machine learning
Blocking improves locality

```c
double *c = (double *) calloc(sizeof(double), n*n);
/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                c[i*n+j1] += a[i*n+k1]*b[k1*n + j1];
}
```
PhiPac/ATLAS: MMM Generator

source: Pingali, Yotov, Cornell U.
FFTW: Discrete Fourier Transform (DFT)

Frigo, Johnson

**Installation**
configure/make

**Usage**
\[ d = \text{dft}(n) \]
\[ d(x, y) \]

Twiddles

Search for fastest computation strategy

- **n = 1024**
  - **radix 16**
  - **16 base case**
  - **64 base case**
- **radix 8**
  - **8 base case**
  - **8 base case**

FFTW: Codelet Generator

Frigo

\[ n \]
\[ \text{DFT codelet generator} \]
\[ \text{dft}_n(*x, *y, \ldots) \]

*fixed size DFT functions*

*straightline code*
OSKI: Sparse Matrix-Vector Multiplication

*Vuduc, Im, Yelick, Demmel*

- **Blocking for registers:**
  - Improves locality (reuse of input vector)
  - But creates overhead (zeros in block)
OSKI: Sparse Matrix-Vector Multiplication

Gain by blocking (dense MVM)

Overhead by blocking

\[
\frac{16}{9} = 1.77
\]

1.4/1.77 = 0.79 (no gain)

search used
no search used

FFT codelet generator

OSKI sparse MVM

FFTW adaptive library

time of implementation

time of installation
platform known

time of use
problem parameters known
Overview

- Rough classification of autotuning efforts seen in course
- Use of machine learning I [de Mesmay et al., IPDPS 2010]
**Online tuning**
(time of use)

**Installation**
configure/make

**Use**
\[ d = \text{dft}(n) \]
\[ d(x,y) \]

**Twiddles**
Search for fastest computation strategy

**Offline tuning**
(time of installation)

**Installation**
configure/make

**Use**
\[ d = \text{dft}(n) \]
\[ d(x,y) \]

**Twiddles**

**Goal**

---

**Library Structure: Examples**

**DFT: scalar code**

**DFT: full-fledged (vectorized and parallel code)**

OpenMP loop of scaled dfts
Library Structure: Examples

DFT: scalar code

Recursive choice:

Example selections for $n = 16$:

- $n = 2^k$
  - base case?
  - radix?

- $n = 16$
  - no base case
  - radix 4

  - $4$
    - base case
    - $2$
      - base case
      - base case

DFT: full-fledged (vectorized and parallel code)

Recursive choice:

Example selections for $n = 1024$:

- $n = 2^k$
  - base case?
  - radix?
  - threading?
  - #threads?
  - twiddles?
  - loop exchange?

- $n = 1024$
  - no base case
  - radix 16
  - threading!
  - 4 threads
  - twiddles on the fly
  - no loop exchange

  - $64$
    - base case
    - $8$
      - base case
      - base case
Upon installation, generate decision trees for each choice

Example:
```java
if ( n <= 65536 ) {
    if ( n <= 32 ) {
        if ( n <= 4 ) { return 2; }
        else { return 4; }
    } else {
        if ( n <= 1024 ) {
            if ( n <= 256 ) { return 9; }
            else { return 32; }
        } else {
            ***************
        }
    }
}
```

Statistical Classification: C4.5

### Features (events)

<table>
<thead>
<tr>
<th>Outlook</th>
<th>Temperature</th>
<th>Humidity</th>
<th>Windy</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>sunny</td>
<td>85</td>
<td>85</td>
<td>false</td>
<td>don’t play</td>
</tr>
<tr>
<td>sunny</td>
<td>80</td>
<td>90</td>
<td>true</td>
<td>play</td>
</tr>
<tr>
<td>overcast</td>
<td>83</td>
<td>78</td>
<td>false</td>
<td>play</td>
</tr>
<tr>
<td>rain</td>
<td>70</td>
<td>96</td>
<td>false</td>
<td>play</td>
</tr>
<tr>
<td>rain</td>
<td>68</td>
<td>80</td>
<td>false</td>
<td>play</td>
</tr>
<tr>
<td>rain</td>
<td>65</td>
<td>70</td>
<td>true</td>
<td>play</td>
</tr>
<tr>
<td>overcast</td>
<td>64</td>
<td>65</td>
<td>true</td>
<td>play</td>
</tr>
<tr>
<td>sunny</td>
<td>72</td>
<td>95</td>
<td>false</td>
<td>play</td>
</tr>
<tr>
<td>sunny</td>
<td>69</td>
<td>70</td>
<td>false</td>
<td>play</td>
</tr>
<tr>
<td>sunny</td>
<td>75</td>
<td>80</td>
<td>false</td>
<td>play</td>
</tr>
<tr>
<td>sunny</td>
<td>75</td>
<td>70</td>
<td>true</td>
<td>play</td>
</tr>
<tr>
<td>overcast</td>
<td>72</td>
<td>90</td>
<td>true</td>
<td>play</td>
</tr>
<tr>
<td>overcast</td>
<td>81</td>
<td>75</td>
<td>false</td>
<td>play</td>
</tr>
<tr>
<td>rain</td>
<td>71</td>
<td>80</td>
<td>true</td>
<td>don’t play</td>
</tr>
</tbody>
</table>

P(play|windy=false) = 6/8
P(don’t play|windy=false) = 2/8
P(play|windy=true) = 1/2
P(don’t play|windy=false) = 1/2

H(windy=false) = 0.81
H(windy=true) = 1.0
H(windy) = 0.89
H(outlook) = 0.69
H(humidity) = ...

Entropy of Features
Application to Libraries

- Features = arguments of functions (except variable pointers)
  
  \[
  \text{dft}(\text{int } n, \text{ cpx } *y, \text{ cpx } *x) \\
  \text{dft\_strided}(\text{int } n, \text{ int } \text{istr}, \text{ cpx } *y, \text{ cpx } *x) \\
  \text{dft\_scaled}(\text{int } n, \text{ int } \text{str}, \text{ cpx } *d, \text{ cpx } *y, \text{ cpx } *x)
  \]

- At installation time:
  - Run search for a few input sizes \( n \)
  - Yields training set: features and associated decisions (several for each size)
  - Generate decision trees using C4.5 and insert into library

Experimental Setup

- 3GHz Intel Xeon 5160 (2 Core 2 Duos = 4 cores)
- Linux 64-bit, icc 10.1
- Libraries:
  - IPP 5.3
  - FFTW 3.2 alpha 2
  - Spiral-generated library

![Graph showing performance comparison between different libraries](image-url)
“All” Sizes

- All sizes $n \leq 2^{18}$, with prime factors $\leq 19$
“All” Sizes

- All sizes \( n \leq 2^{18} \), with prime factors \( \leq 19 \)
- Higher order fit of all sizes

Message of Lecture

- Machine learning should be used in autotuning
  - Overcomes the problem of expensive searches
  - Relatively easy to do
  - Applicable to any search-based approach
  - Removes searches or better searches