Optimizing FFT-based Polynomial Arithmetic for Data Locality and Parallelism

Marc Moreno Maza

University of Western Ontario, Canada

MaGiX@LIX Workshop
September 20, 2011
**Introduction: code optimization**

**Optimizing for data locality**
- Computer cache memories have led to introduce a new complexity measure for algorithms and new performance counters for code.
- Optimizing for data locality brings large speedup factors, as we shall see.
Introduction: code optimization

Optimizing for data locality

- Computer cache memories have led to introduce a new complexity measure for algorithms and new performance counters for code.
- Optimizing for data locality brings large speedup factors, as we shall see.

Optimizing for parallelism

- All recent home and office desktops/laptops are parallel machines; moreover “GPU cards bring supercomputing to the masses,” (NVIDIA moto).
- Optimizing for parallelism improves the use of computing resources (Green!)
- And optimizing for data locality is often a first step!
Introduction: code optimization

Optimizing for data locality
- Computer cache memories have led to introduce a new complexity measure for algorithms and new performance counters for code.
- Optimizing for data locality brings large speedup factors, as we shall see.

Optimizing for parallelism
- All recent home and office desktops/laptops are parallel machines; moreover “GPU cards bring supercomputing to the masses,” (NVIDIA moto).
- Optimizing for parallelism improves the use of computing resources (Green!)
- And optimizing for data locality is often a first step!

Optimizing for algebraic complexity in this context
- Consider a 1-level cache machine with a $Z$-word cache and $L$-word cache lines.
- Consider a polynomial/matrix operation running within $n^\alpha$ coefficient operations, up to a small constant say 2 to 10.
- A typical naive implementation will incur $n^\alpha/L$ cache misses, which reduce to $n^\alpha/(\sqrt{ZL})$ for a cache-friendly algorithm.
- Moreover, execution and memory models (say multicore vs manycore) have an impact on algorithm design.
Multicores

- **Cache coherency circuitry** operate at higher rate than off-chip.
- Cores on a multi-core implement the *same architecture features as single-core systems* such as instruction pipeline parallelism (ILP), vector-processing, hyper-threading.
- Two processing cores sharing the same bus and memory bandwidth may limit performances.
- High levels of *false or true sharing* and synchronization can easily overwhelm the advantage of parallelism.
**Introduction: hardware**

**Multicores**
- **Cache coherency circuitry** operate at higher rate than off-chip.
- Cores on a multi-core implement the **same architecture features as single-core systems** such as instruction pipeline parallelism (ILP), vector-processing, hyper-threading.
- Two processing cores sharing the same bus and memory bandwidth may limit performances.
- High levels of **false or true sharing** and synchronization can easily overwhelm the advantage of parallelism.

**Manycores**
- Hardware allocates resources to thread blocks and schedules threads, thus **no parallelization overhead**, contrary to multicores.
- No synchronization possible between thread blocks, which force to think differently, but which provides **automatic scaling as long as enough parallelism is exposed**.
- Shared memories and global memory offer **a form of CRCW**.
- **Shared memories are tiny and streaming processors have very limited architecture features**, contrary to the cores in a multicore.
Typical algorithms have high algebraic complexity, say $n^\alpha$ for $\alpha > 1$ and low span, say $\log^\beta(n)$ for some $\beta \geq 1$. Thus, a lot of parallelism opportunities, at least in theory.
Typical algorithms have high algebraic complexity, say $n^\alpha$ for $\alpha > 1$ and low span, say $\log^\beta(n)$ for some $\beta \geq 1$. Thus, a lot of parallelism opportunities, at least in theory.

Except some \{french empècheur de tourner en rond\}, say the Euclidean Algorithm over $\mathbb{Z}$.
Typical algorithms have high algebraic complexity, say $n^\alpha$ for $\alpha > 1$ and low span, say $\log^\beta(n)$ for some $\beta \geq 1$. Thus, a lot of parallelism opportunities, at least in theory.

Except some \textit{french empècheur de tourner en rond}, say the Euclidean Algorithm over $\mathbb{Z}$.

As mentioned before, the algebraic-complexity-to-cache-complexity ratio is often a constant: bad!
Typical algorithms have high algebraic complexity, say $n^\alpha$ for $\alpha > 1$ and low span, say $\log^\beta(n)$ for some $\beta \geq 1$. Thus, a lot of parallelism opportunities, at least in theory.

Except some \textit{\texttt{\textbackslash french emp\`echeur de tourner en rond}}, say the Euclidean Algorithm over $\mathbb{Z}$.

As mentioned before, the algebraic-complexity-to-cache-complexity ratio is often a constant: bad!

Unless efforts are made to make algorithms cache optimal.
Typical algorithms have high algebraic complexity, say $n^\alpha$ for $\alpha > 1$ and low span, say $\log^\beta(n)$ for some $\beta \geq 1$. Thus, a lot of parallelism opportunities, at least in theory.

Except some \{\textit{french empècheur de tourner en rond}\}, say the Euclidean Algorithm over $\mathbb{Z}$.

As mentioned before, the algebraic-complexity-to-cache-complexity ratio is often a constant: bad!

Unless efforts are made to make algorithms cache optimal.

Polynomial/matrix algorithms are often divide-and-conquer which helps avoiding data access competition among threads.
Typical algorithms have high algebraic complexity, say $n^\alpha$ for $\alpha > 1$ and low span, say $\log^\beta(n)$ for some $\beta \geq 1$. Thus, a lot of parallelism opportunities, at least in theory.

Except some \textit{french empècheur de tourner en rond}, say the Euclidean Algorithm over $\mathbb{Z}$.

As mentioned before, the algebraic-complexity-to-cache-complexity ratio is often a constant: bad!

Unless efforts are made to make algorithms cache optimal.

Polynomial/matrix algorithms are often divide-and-conquer which helps avoiding data access competition among threads.

Of course, these lock-free approaches increase the span but so do mutexes anyway!
1. Hierarchical memories and cache complexity

2. Balanced polynomial arithmetic on multicores

3. Bivariate polynomial systems on the GPU

4. Status of our libraries
Plan

1. Hierarchical memories and cache complexity
2. Balanced polynomial arithmetic on multicores
3. Bivariate polynomial systems on the GPU
4. Status of our libraries
Hierarchical memories and cache complexity

The \((Z, L)\) ideal cache model (1/2)

- The ideal (data) cache of \(Z\) words partitioned into \(Z/L\) cache lines.
- Data moved between cache and main memory are always cache lines.
- The cache is **tall**, that is, \(Z\) is much larger than \(L\), say \(Z \in \Omega(L^2)\).
- The processor can only reference words that reside in the cache.

![Figure 1: The ideal-cache model](image-url)
The \((Z, L)\) ideal cache model (2/2)

- If the CPU refers to a word not in cache, a **cache miss** occurs.
- The ideal cache is **fully associative**: cache lines can be stored anywhere in the cache.
- The ideal cache uses the **optimal off-line strategy of replacing** the cache line whose next access is furthest in the future.

**Figure 1:** The ideal-cache model
Hierarchical memories and cache complexity

A typical naive matrix multiplication C code

```c
#define IND(A, x, y, d) A[(x)*(d)+(y)]

uint64_t testMM(const int x, const int y, const int z)
{
    double *A; double *B; double *C; double *Cx;
    long started, ended;
    float timeTaken;
    int i, j, k;
    srand(getSeed());
    A = (double *)malloc(sizeof(double)*x*y);
    B = (double *)malloc(sizeof(double)*x*z);
    C = (double *)malloc(sizeof(double)*y*z);
    for (i = 0; i < x*z; i++) B[i] = (double) rand();
    for (i = 0; i < y*z; i++) C[i] = (double) rand();
    for (i = 0; i < x*y; i++) A[i] = 0 ;
    started = example_get_time();
    for (i = 0; i < x; i++)
        for (j = 0; j < y; j++)
            for (k = 0; k < z; k++)
                // A[i][j] += B[i][k] + C[k][j];
                IND(A,i,j,y) += IND(B,i,k,z) * IND(C,k,j,z);
    ended = example_get_time();
    timeTaken = (ended - started)/1.f;
    return timeTaken;
}
```
Analyzing cache misses in the naive and transposed multiplication

Let \( A, B \) and \( C \) have format \((m, n), (m, p)\) and \((p, n)\) respectively.

\( A \) is scanned one, so \( \frac{mn}{L} \) cache misses if \( L \) is the number of coefficients per cache line.

\( B \) is scanned \( n \) times, so \( \frac{mnp}{L} \) cache misses if the cache cannot hold a row.

\( C \) is accessed “nearly randomly” (for \( m \) large enough) leading to \( mnp \) cache misses.

Since \( 2mnp \) arithmetic operations are performed, this means roughly one cache miss per flop!

If \( C \) is transposed, then the ratio improves to 1-for-\( L \).
Transposing for optimizing spatial locality

```c
float testMM(const int x, const int y, const int z)
{
    double *A; double *B; double *C; double *Cx;
    long started, ended; float timeTaken; int i, j, k;
    A = (double *)malloc(sizeof(double)*x*y);
    B = (double *)malloc(sizeof(double)*x*z);
    C = (double *)malloc(sizeof(double)*y*z);
    Cx = (double *)malloc(sizeof(double)*y*z);
    srand(getSeed());
    for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
    for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
    for (i = 0; i < x*y; i++) A[i] = 0 ;
    started = example_get_time();
    for(j =0; j < y; j++)
        for(k=0; k < z; k++)
            IND(Cx,j,k,z) = IND(C, k, j, y);
    for (i = 0; i < x; i++)
        for (j = 0; j < y; j++)
            for (k = 0; k < z; k++)
                IND(A, i, j, y) += IND(B, i, k, z) *IND(Cx, j, k, z);
    ended = example_get_time();
    timeTaken = (ended - started)/1.f;
    return timeTaken;
}
```
Hierarchical memories and cache complexity

Analyzing cache misses in the tiled multiplication

Let $A$, $B$ and $C$ have format $(m, n)$, $(m, p)$ and $(p, n)$ respectively.
Assume all tiles are square of order $B$ and three fit in cache.
If $C$ is transposed, then loading three blocks in cache cost $3B^2/L$.
This process happens $n^3/B^3$ times, leading to $3n^3/(BL)$ cache misses.
Three blocks fit in cache for $3B^2 < Z$, if $Z$ is the cache size.
So $O(n^3/(\sqrt{ZL}))$ cache misses, if $B$ is well chosen, which is optimal.
float testMM(const int x, const int y, const int z)
{
    double *A; double *B; double *C; double *Cx;
    long started, ended; float timeTaken; int i, j, k, i0, j0, k0;
    A = (double *)malloc(sizeof(double)*x*y);
    B = (double *)malloc(sizeof(double)*x*z);
    C = (double *)malloc(sizeof(double)*y*z);
    srand(getSeed());
    for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
    for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
    for (i = 0; i < x*y; i++) A[i] = 0 ;
    started = example_get_time();
    for (i = 0; i < x; i += BLOCK_X)
        for (j = 0; j < y; j += BLOCK_Y)
            for (k = 0; k < z; k += BLOCK_Z)
                for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
                    for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
                        for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
                            IND(A,i0,j0,y) += IND(B,i0,k0,z) * IND(C,j0,k0,z);
    ended = example_get_time();
    timeTaken = (ended - started)/1.f;
    return timeTaken;
}
### Experimental results

Computing the product of two $n \times n$ matrices on my laptop (Core2 Duo CPU P8600 @ 2.40GHz, L1 cache of 3072 KB, 4 GBytes of RAM)

<table>
<thead>
<tr>
<th>$n$</th>
<th>naive</th>
<th>transposed</th>
<th>speedup</th>
<th>64 × 64-tiled</th>
<th>speedup</th>
<th>t. &amp; t.</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>7</td>
<td>3</td>
<td>speedup</td>
<td>7</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>26</td>
<td>43</td>
<td>6.81</td>
<td>1928</td>
<td>0.936</td>
<td>23</td>
<td>9.65</td>
</tr>
<tr>
<td>512</td>
<td>1805</td>
<td>265</td>
<td>155</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>24723</td>
<td>3730</td>
<td>6.62</td>
<td>14020</td>
<td>1.76</td>
<td>1490</td>
<td>16.59</td>
</tr>
<tr>
<td>2048</td>
<td>271446</td>
<td>29767</td>
<td>9.11</td>
<td>112298</td>
<td>2.41</td>
<td>11960</td>
<td>22.69</td>
</tr>
<tr>
<td>4096</td>
<td>2344594</td>
<td>238453</td>
<td>9.83</td>
<td>1009445</td>
<td>2.32</td>
<td>101264</td>
<td>23.15</td>
</tr>
</tbody>
</table>

Timings are in milliseconds.

- The cache-oblivious multiplication (more on this later) runs within 12978 and 106758 for $n = 2048$ and $n = 4096$ respectively.
- More optimization tricks can be used, such as using vector parallelism (SSE instructions).
- Optimized C implementation of Strassen and Waksman algorithms are at least one order of magnitude. Special thanks to Nazul Islam (UW).
Hierarchical memories and cache complexity

Other performance counters

Hardware count events

- **CPI (Clock cycles Per Instruction):** the number of clock cycles that happen when an instruction is being executed. With pipelining we can improve the CPI by exploiting instruction level parallelism.

- **L1 and L2 Cache Miss Rate.**

- **Instructions Retired:** In the event of a misprediction, instructions that were scheduled to execute along the mispredicted path must be canceled.

<table>
<thead>
<tr>
<th></th>
<th>CPI</th>
<th>L1 Miss Rate</th>
<th>L2 Miss Rate</th>
<th>Percent SSE</th>
<th>Instructions Retired</th>
</tr>
</thead>
<tbody>
<tr>
<td>In C</td>
<td>4.78</td>
<td>0.24</td>
<td>0.02</td>
<td>43%</td>
<td>13,137,280,000</td>
</tr>
<tr>
<td>Transposed</td>
<td>1.13</td>
<td>0.15</td>
<td>0.02</td>
<td>50%</td>
<td>13,001,486,336</td>
</tr>
<tr>
<td>Tiled</td>
<td>0.49</td>
<td>0.02</td>
<td>0</td>
<td>39%</td>
<td>18,044,811,264</td>
</tr>
</tbody>
</table>
A matrix transposition cache-oblivious and cache-optimal algorithm

- Given an $m \times n$ matrix $A$ stored in a row-major layout, compute and store $A^T$ into an $n \times m$ matrix $B$ also stored in a row-major layout.
- A naive approach would incur $O(mn)$ cache misses, for $n, m$ large enough.
- The algorithm $\text{Rec-Transpose}$ below incurs $\Theta(1 + mn/L)$ cache misses, which is optimal.
  1. If $n \geq m$, the $\text{Rec-Transpose}$ algorithm partitions
     \[
     A = (A_1 \quad A_2), \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}
     \]
     and recursively executes $\text{Rec-Transpose}(A_1, B_1)$ and $\text{Rec-Transpose}(A_2, B_2)$.
  2. If $m > n$, the $\text{Rec-Transpose}$ algorithm partitions
     \[
     A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix}, \quad B = (B_1 \quad B_2)
     \]
     and recursively executes $\text{Rec-Transpose}(A_1, B_1)$ and $\text{Rec-Transpose}(A_2, B_2)$. 
void DC_matrix_transpose(int *A, int lda, int i0, int i1, int j0, int dj0, int j1 /*, int dj1 = 0 */){
    const int THRESHOLD = 16; // tuned for the target machine

tail:
    int di = i1 - i0, dj = j1 - j0;
    if (dj >= 2 * di && dj > THRESHOLD) {
        int dj2 = dj / 2;
        cilk_spawn DC_matrix_transpose(A, lda, i0, i1, j0, dj0, j0 + dj2);
        j0 += dj2; dj0 = 0; goto tail;
    } else if (di > THRESHOLD) {
        int di2 = di / 2;
        cilk_spawn DC_matrix_transpose(A, lda, i0, i0 + di2, j0, dj0, j1);
        i0 += di2; j0 += dj0 * di2; goto tail;
    } else {
        for (int i = i0; i < i1; ++i) {
            for (int j = j0; j < j1; ++j) {
                int x = A[j * lda + i];
                A[j * lda + i] = A[i * lda + j];
                A[i * lda + j] = x;
            }
            j0 += dj0;
        }
    }
}
### Cache-oblivious matrix transposition works in practice!

<table>
<thead>
<tr>
<th>size</th>
<th>Naive</th>
<th>Cache-oblivious</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000×5000</td>
<td>126</td>
<td>79</td>
<td>1.59</td>
</tr>
<tr>
<td>10000×10000</td>
<td>627</td>
<td>311</td>
<td>2.02</td>
</tr>
<tr>
<td>20000×20000</td>
<td>4373</td>
<td>1244</td>
<td>3.52</td>
</tr>
<tr>
<td>30000×30000</td>
<td>23603</td>
<td>2734</td>
<td>8.63</td>
</tr>
<tr>
<td>40000×40000</td>
<td>62432</td>
<td>4963</td>
<td>12.58</td>
</tr>
</tbody>
</table>

- **Intel(R) Xeon(R) CPU E7340 @ 2.40GHz**
- L1 data 32 KB, L2 4096 KB, cache line size 64bytes
- **Both codes run on 1 core** on a node with 128GB.
- The ration comes simply from an optimal memory access pattern.
A cache-oblivious matrix multiplication algorithm

To multiply an $m \times n$ matrix $A$ and an $n \times p$ matrix $B$, the Rec-Mult algorithm halves the largest of the three dimensions and recurs according to one of the following three cases:

\[
\begin{align*}
(A_1 & A_2) \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = (A_1 B & A_2 B), \quad (1) \\
(A_1 & A_2) \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2, \quad (2) \\
A \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = (AB_1 & AB_2). \quad (3)
\end{align*}
\]

- In case (1), we have $m \geq \max \{n, p\}$. Matrix $A$ is split horizontally, and both halves are multiplied by matrix $B$.
- In case (2), we have $n \geq \max \{m, p\}$. Both matrices are split, and the two halves are multiplied.
- In case (3), we have $p \geq \max \{m, n\}$. Matrix $B$ is split vertically, and each half is multiplied by $A$.
- The base case occurs when $m = n = p = 1$.
- The algorithm Rec-Mult above incurs
  \[
  \Theta(m + n + p + (mn + np + mp)/L + mnp/(L\sqrt{Z}))
  \] cache misses, which is optimal.
The ideal cache model and cache complexity, despite of their strong assumptions, are **practically verified** in most cases I have studied.

Cache complexity improvements can be verified in practice **even on algorithms whose algebraic complexity is linear**: transposition, counting sort.

Cache-naive plain univariate polynomial multiplication incurs $\Theta(n^2/L)$ cache misses while cache-optimal plain univariate polynomial multiplication incurs only $\Theta(n^2/(ZL))$ cache misses.

However this latter algorithm is **tricky to implement efficiently** and I am not happy yet with my experimental results.
Plan

1. Hierarchical memories and cache complexity

2. Balanced polynomial arithmetic on multicores

3. Bivariate polynomial systems on the GPU

4. Status of our libraries
Background

- Computing 1D FFTs of size 1000 or less is common.
- For those, there is not enough work to obtain good speedup.
- In addition, we have obtained over the years highly optimized serial C code for 1D FFTs (based on TFT techniques)

Assumptions and goals

- 1-D FFTs are computed by a black box program which could be serial code.
- We want FFT-based dense multivariate arithmetic routines that are cache friendly and targeting multicores.
Let $\mathbb{K}$ be a finite field and $f, g \in \mathbb{K}[x_1 < \cdots < x_n]$ be polynomials with $n \geq 2$.

Define $d_i = \deg(f, x_i)$ and $d'_i = \deg(g, x_i)$, for all $i$.

Assume there exists a primitive $s_i$-th root of unity $\omega_i \in \mathbb{K}$, for all $i$, where $s_i$ is a power of 2 satisfying $s_i \geq d_i + d'_i + 1$.

Then $fg$ can be computed as follows.

**Step 1.** Evaluate $f$ and $g$ at each point $P$ (i.e. $f(P), g(P)$) of the $n$-dimensional grid $((\omega_1^{e_1}, \ldots, \omega_n^{e_n}), 0 \leq e_1 < s_1, \ldots, 0 \leq e_n < s_n)$ via $n$-D FFT.

**Step 2.** Evaluate $fg$ at each point $P$ of the grid, simply by computing $f(P)g(P)$,

**Step 3.** Interpolate $fg$ (from its values on the grid) via $n$-D FFT.
Speedup factors of bivariate interpolation \((d_1 = d_2)\)

![Graph showing speedup factors of bivariate interpolation for different numbers of cores.](image)

Special thanks to Matteo Frigo for his cache-efficient code for matrix transposition!
Speedup factors of bivariate multiplication \((d_1 = d_2 = d'_1 = d'_2)\)
Challenges: irregular input data

Balanced polynomial arithmetic on multicores

Number of Cores

Speedup

- linear speedup
- bivariate (32765, 63)
- 8-variate (all 4)
- 4-variate (1023, 1, 1, 1023)
- univariate (25427968)
## Performance analysis with VTune

<table>
<thead>
<tr>
<th>No.</th>
<th>Size of Two Input Polynomials</th>
<th>Product Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$8191 \times 8191$</td>
<td>$268402689$</td>
</tr>
<tr>
<td>2</td>
<td>$259575 \times 258$</td>
<td>$268401067$</td>
</tr>
<tr>
<td>3</td>
<td>$63 \times 63 \times 63 \times 63$</td>
<td>$260144641$</td>
</tr>
<tr>
<td>4</td>
<td>8 vars. of deg. 5</td>
<td>$214358881$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No.</th>
<th>INST RETIRED. ANY $\times 10^9$</th>
<th>Clocks per Instruction Retired</th>
<th>L2 Cache Miss Rate $(\times 10^{-3})$</th>
<th>Modified Data Sharing Ratio $(\times 10^{-3})$</th>
<th>Time on 8 Cores (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>659.555</td>
<td>0.810</td>
<td>0.333</td>
<td>0.078</td>
<td>16.15</td>
</tr>
<tr>
<td>2</td>
<td>713.882</td>
<td>0.890</td>
<td>0.735</td>
<td>0.192</td>
<td>19.52</td>
</tr>
<tr>
<td>3</td>
<td>714.153</td>
<td>0.854</td>
<td>1.096</td>
<td>0.635</td>
<td>22.44</td>
</tr>
<tr>
<td>4</td>
<td>1331.340</td>
<td>1.418</td>
<td>1.177</td>
<td>0.576</td>
<td>72.99</td>
</tr>
</tbody>
</table>
Let $s = s_1 \cdots s_n$. The number of operations in $\mathbb{K}$ for computing $fg$ via n-D FFT is
\[
\frac{9}{2} \sum_{i=1}^{n} (\prod_{j \neq i} s_j) s_i \lg(s_i) + (n+1)s = \frac{9}{2} s \lg(s) + (n+1)s.
\]

Under our 1-D FFT black box assumption, the span of \textit{Step 1} is
\[
\frac{9}{2} (s_1 \lg(s_1) + \cdots + s_n \lg(s_n)),
\]
and the parallelism of \textit{Step 1} is lower bounded by
\[
s / \max(s_1, \ldots, s_n).
\] (4)

Let $L$ be the size of a cache line. For some constant $c > 0$, the number of cache misses of \textit{Step 1} is upper bounded by
\[
n \frac{cs}{L} + cs\left(\frac{1}{s_1} + \cdots + \frac{1}{s_n}\right).
\] (5)
Let $Q(s_1, \ldots, s_n)$ denotes the total number of cache misses for the whole algorithm, for some constant $c$ we obtain

\[
redQ(s_1, \ldots, s_n) \leq cs \frac{n + 1}{L} + cs\left(\frac{1}{s_1} + \cdots + \frac{1}{s_n}\right) \tag{6}
\]

Observe we have $\frac{n}{s^{1/n}} \leq \frac{1}{s_1} + \cdots + \frac{1}{s_n}$

When $s_i = s^{1/n}$ holds for all $i$, we have

\[
Q(s_1, \ldots, s_n) \leq ncs\left(\frac{2}{L} + \frac{1}{s^{1/n}}\right) \tag{7}
\]

For $n \geq 2$, Expr. (7) is minimized at $n = 2$ and $s_1 = s_2 = \sqrt{s}$.
Moreover, when $n = 2$, for a fixed $s = s_1s_2$, the parallelism is maximized at $s_1 = s_2 = \sqrt{s}$. 
Balanced multiplication

**Definition.** A pair of bivariate polynomials $p, q \in \mathbb{K}[u, v]$ is **balanced** if
\[
\deg(p, u) + \deg(q, u) = \deg(p, v) + \deg(q, v).
\]

**Algorithm.** Let $f, g \in \mathbb{K}[x_1 < \ldots < x_n]$. W.l.o.g. one can assume $d_1 \gg d_i$ and $d_1' \gg d_i$ for $2 \leq i \leq n$ (up to variable re-ordering and contraction). Then we obtain $f_b g_b \in \mathbb{K}[u, v]$ by

1. **Step 1.** Inverse Kronecker substitution $x_1$ to $\{u, v\}$

2. **Step 2.** Direct Kronecker substitution $\{v, x_2, \ldots, x_n\}$ to $v$.

such that

- the pair $f_b, g_b$ is (nearly) a balanced pair and $f_b g_b$ has dense size at most twice that of $fg$.
- we can recover the product $fg$ from the product $f_b g_b$.

Speedup factors of balanced multiplication \((d_2 = d_2 = d_3 = 2)\)

- Ext. + Contr. of 4-D to 2-D TFT on 1 core (7.6-15.7 s)
- Kronecker substitution of 4-D to 1-D TFT on 1 core (6.8-14.1 s)
- Ext. + Contr. of 4-D to 2-D TFT on 2 cores (1.96x speedup, 1.75x net gain)
- Ext. + Contr. of 4-D to 2-D TFT on 16 cores (7.0-11.3x speedup, 6.2-10.3x net gain)
Bivariate multiplication for input degree range of $[1024, 2048)$ on 1 core.
2-D FFT method on 8 cores (0.806-0.902 s, 7.2-7.3x speedup)
2-D TFT method on 8 cores (0.309-1.08 s, 6.8-7.6x speedup)

Bivariate multiplication for input degree range of $[1024, 2048)$ on 8 cores.
Bivariate multiplication for input degree range of $[1024, 2048)$ on 16 cores. **Question:** why TFT always beats FFT on 16 cores?
Balanced polynomial arithmetic on multicores

Summary and notes

- Balanced data traversal provides work load balancing.
- But more importantly it minimizes cache misses and thus helps reducing memory traffic.
- Other operations can be balanced: normal form computations and subresultant chain computation.
- And yes, considering fast polynomial arithmetic independently of data locality and parallelism makes no sense today!
Plan

1. Hierarchical memories and cache complexity

2. Balanced polynomial arithmetic on multicores

3. Bivariate polynomial systems on the GPU

4. Status of our libraries
Background

- No parallelization overheads on the GPU since the hardware schedules the threads.
- Most FFTs on GPUs are for floats, such as the NVIDIA CUFFT library.
- What about finite fields?

Testing in GB/s

<table>
<thead>
<tr>
<th>(\log_2 n)</th>
<th>memset</th>
<th>Main Mem to GPU</th>
<th>GPU to Main Mem</th>
<th>GPU Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>1.56</td>
<td>1.33</td>
<td>1.52</td>
<td>61.6</td>
</tr>
<tr>
<td>24</td>
<td>1.56</td>
<td>1.34</td>
<td>1.52</td>
<td>69.9</td>
</tr>
<tr>
<td>25</td>
<td>1.39</td>
<td>1.35</td>
<td>1.53</td>
<td>75.0</td>
</tr>
<tr>
<td>26</td>
<td>1.39</td>
<td>1.28</td>
<td>1.50</td>
<td>77.4</td>
</tr>
<tr>
<td>27</td>
<td>1.43</td>
<td>1.35</td>
<td>1.49</td>
<td>79.0</td>
</tr>
</tbody>
</table>

- Intel Core 2 Quad Q9400 @ 2.66GHz, 6GB memory, memory interface width 128 bits
- GeForce GTX 285, 1GB global memory, \(30 \times 8\) cores, memory interface
Extract parallelism from structural formulas

$I_n \otimes A$: block parallelism

$I_4 \otimes \text{DFT}_2 =
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1
\end{bmatrix}

Extract parallelism from structural formulas

\[ A \otimes I_n: \text{vector parallelism} \]

\[ \text{DFT}_2 \otimes I_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \end{bmatrix} \]
Stockham FFT

\[
\text{DFT}_{2^k} = \prod_{i=0}^{k-1} (\text{DFT}_2 \otimes I_{2^{k-1}})(D_{2, 2^{k-i-1}} \otimes I_{2^i})(L_{2}^{2^{k-i}} \otimes I_{2^i})
\]

void stockham_dev(int *X_d, int n, int k, const int *W_d, int p)
{
    int *Y_d;
    cudaMalloc((void **)&Y_d, sizeof(int) * n);
    butterfly_dev(Y_d, X_d, k, p);
    for (int i = k - 2; i >= 0; --i) {
        stride_transpose2_dev(X_d, Y_d, k, i);
        stride_twiddle2_dev(X_d, W_d, k, i, p);
        butterfly_dev(Y_d, X_d, k, p);
    }
    cudaMemcpy(X_d, Y_d, sizeof(int)*n, cudaMemcpyDeviceToDevice);
    cudaFree(Y_d);
}
Cooley-Tukey FFT

\[ \text{DFT}_{2^k} = \left( \prod_{i=1}^{k} (I_{2i-1} \otimes \text{DFT}_2 \otimes I_{2^{k-i}}) T_{n,i} \right) R_n \]

with the twiddle factor matrix \( T_{n,i} = I_{2i-1} \otimes D_{2,2^{k-i}} \) and the bit-reversal permutation matrix

\[ R_n = (I_{n/2} \otimes L_2^2)(I_{n/2^2} \otimes L_2^4) \cdots (I_1 \otimes L_2^n). \]
### Timing FFT in milliseconds

<table>
<thead>
<tr>
<th>e</th>
<th>modpn</th>
<th>Cooley-Tukey</th>
<th>C-T + Mem</th>
<th>Stockham</th>
<th>S + Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td>ratio</td>
<td>time</td>
<td>ratio</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
<td>0.5</td>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>1</td>
<td>3.0</td>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>2</td>
<td>2.0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>3</td>
<td>3.3</td>
<td>3</td>
<td>3.3</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
<td>4</td>
<td>4.0</td>
<td>5</td>
<td>3.2</td>
</tr>
<tr>
<td>18</td>
<td>37</td>
<td>6</td>
<td>6.2</td>
<td>9</td>
<td>4.1</td>
</tr>
<tr>
<td>19</td>
<td>71</td>
<td>11</td>
<td>6.5</td>
<td>15</td>
<td>6.5</td>
</tr>
<tr>
<td>20</td>
<td>174</td>
<td>22</td>
<td>7.9</td>
<td>28</td>
<td>6.2</td>
</tr>
<tr>
<td>21</td>
<td>470</td>
<td>44</td>
<td>10.7</td>
<td>56</td>
<td>8.4</td>
</tr>
<tr>
<td>22</td>
<td>997</td>
<td>83</td>
<td>12.0</td>
<td>105</td>
<td>9.5</td>
</tr>
<tr>
<td>23</td>
<td>2070</td>
<td>165</td>
<td>12.5</td>
<td>210</td>
<td>9.9</td>
</tr>
<tr>
<td>24</td>
<td>4194</td>
<td>330</td>
<td>12.7</td>
<td>418</td>
<td>10.0</td>
</tr>
<tr>
<td>25</td>
<td>8611</td>
<td>667</td>
<td>12.9</td>
<td>842</td>
<td>10.2</td>
</tr>
<tr>
<td>26</td>
<td>17617</td>
<td>1338</td>
<td>13.2</td>
<td>1686</td>
<td>10.4</td>
</tr>
</tbody>
</table>

The GPU is GTX 285.
Solving polynomial systems with GPU support

Main idea

Solving $P(x, y) = Q(x, y) = 0$ is essentially done as follows:

1. Determine necessary conditions on $x$ for $P(x)(y)$ and $Q(x)(y)$ to have common roots; such $x$’s are roots of the resultant $R(x)$ of $P, Q$ w.r.t. $y$.

2. For $x = x_0$ such that $x_0$ is a root of $R$ determine the common solutions of $P(x_0)(y) = 0$ and $Q(x_0)(y) = 0$; this is essentially a GCD computation.

Both steps can be easily done in one **Subresultant Chain Computation**
Subresultant chain computation

\[ P, Q \in \mathbb{Z}_p[x_1, \ldots, x_n, y] \quad \text{Direct computation} \quad \text{subres}(P, Q, y) \in \mathbb{Z}_p[x_1, \ldots, x_n, y] \]

\[ \text{Kronecker’s substitution} \quad \text{Inverse Kronecker} \]

\[ F, G \in \mathbb{Z}_p[x, y] \quad \text{subres}(F(x, y), G(x, y), y) \]

\[ \text{Random translation } \phi_a \quad \text{Inverse } \phi_a \]

\[ F', G' \in \mathbb{Z}_p[x, y] \quad \text{exit, if several random choices } a \text{ failed} \quad \text{subres}(F'(x, y), G'(x, y), y) \]

\[ \text{FFT} \quad \text{Inverse FFT} \]

\[ F'(\omega^i, y), G'(\omega^i, y) \in \mathbb{Z}_p[y] \quad \text{Brown’s algorithm} \quad \text{subres}(F'(\omega^i, y), G'(\omega^i, y), y) \]
**Subresultant chain by evaluation/interpolation**

**Issues with different strategies**

- FFT based technique. Sticky points:
  - Fourier prime limitation
  - valid grid construction
- Subproduct tree technique: a backup solution . . .

**FFT scube on the GPU. Two approaches:**

- Coarse-grained construction:
  - each thread computes a specialized subresultant chain.
  - Low parallelism, but always works.
- Fine-grained construction:
  - Assumes all specialized subresultant chains have the same degree sequence
  - Parallelize the pseudo-divisions
  - Each thread block does a bunch of pieces of pseudo-divisions.
Profiling coarse-grained implementation

- memcpy
- double Expand Ker
- expand_to_list_fft_ker
- list_butterfly_ker
- list_stride_transpose2a_ker
- list_stride_twiddle2a_ker
- list_stride_twiddle2b_ker
- list_stride_transpose2b_ker
- memcpyDtoD_aligned
- transpose_ker
- reset_ker
- subres_chain_tri_ker

Fit In Window: Yes, Max Bar Width Displayed: 1272
Show CPU Time: No, Start Timestamp at Zero: Yes
Profiling fine-grained implementation
## Computing resultants

<table>
<thead>
<tr>
<th>$d$</th>
<th>$t_0$</th>
<th>$t_1$</th>
<th>$t_1/t_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>0.23</td>
<td>0.29</td>
<td>1.3</td>
</tr>
<tr>
<td>40</td>
<td>0.23</td>
<td>0.43</td>
<td>1.9</td>
</tr>
<tr>
<td>50</td>
<td>0.27</td>
<td>1.14</td>
<td>4.2</td>
</tr>
<tr>
<td>60</td>
<td>0.27</td>
<td>1.53</td>
<td>5.7</td>
</tr>
<tr>
<td>70</td>
<td>0.31</td>
<td>3.95</td>
<td>12.7</td>
</tr>
<tr>
<td>80</td>
<td>0.32</td>
<td>4.88</td>
<td>15.3</td>
</tr>
<tr>
<td>90</td>
<td>0.35</td>
<td>5.95</td>
<td>17.0</td>
</tr>
<tr>
<td>100</td>
<td>0.50</td>
<td>19.10</td>
<td>38.2</td>
</tr>
<tr>
<td>110</td>
<td>0.53</td>
<td>17.89</td>
<td>33.8</td>
</tr>
<tr>
<td>120</td>
<td>0.58</td>
<td>19.72</td>
<td>34.0</td>
</tr>
</tbody>
</table>

**Bivariate dense polynomials of total degree** $d$.

<table>
<thead>
<tr>
<th>$d$</th>
<th>$t_0$</th>
<th>$t_1$</th>
<th>$t_1/t_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.23</td>
<td>0.76</td>
<td>3.3</td>
</tr>
<tr>
<td>9</td>
<td>0.24</td>
<td>0.85</td>
<td>3.5</td>
</tr>
<tr>
<td>10</td>
<td>0.25</td>
<td>0.98</td>
<td>3.9</td>
</tr>
<tr>
<td>11</td>
<td>0.24</td>
<td>1.10</td>
<td>4.6</td>
</tr>
<tr>
<td>12</td>
<td>0.30</td>
<td>4.96</td>
<td>16.5</td>
</tr>
<tr>
<td>13</td>
<td>0.31</td>
<td>5.52</td>
<td>17.8</td>
</tr>
<tr>
<td>14</td>
<td>0.32</td>
<td>6.07</td>
<td>19.0</td>
</tr>
<tr>
<td><strong>15</strong></td>
<td><strong>0.78</strong></td>
<td><strong>8.95</strong></td>
<td><strong>11.5</strong></td>
</tr>
<tr>
<td>16</td>
<td>0.65</td>
<td>31.65</td>
<td>48.7</td>
</tr>
<tr>
<td>17</td>
<td>0.66</td>
<td>34.55</td>
<td>52.3</td>
</tr>
<tr>
<td><strong>18</strong></td>
<td><strong>3.46</strong></td>
<td><strong>47.54</strong></td>
<td><strong>13.7</strong></td>
</tr>
<tr>
<td>19</td>
<td>0.73</td>
<td>51.04</td>
<td>69.9</td>
</tr>
<tr>
<td>20</td>
<td>0.75</td>
<td>43.12</td>
<td>57.5</td>
</tr>
</tbody>
</table>

**Trivariate dense polynomials of total degree** $d$.

- $t_0$, GPU fft code
- $t_1$, CPU fft code
- Nvidia Tesla C2050
Bivariate polynomial systems on the GPU

Bivariate solver

![Graph showing comparison between GPU supported bivariate solving and general bivariate solving over partial degrees and time in seconds. The graph illustrates the performance improvement of GPU-supported solving.]
Bivariate polynomial systems on the GPU

Bivariate solver on the CPU

![Graph showing time in seconds vs partial degree for scube construction and total time.](image)
Bivariate polynomial systems on the GPU

Bivariate solver on the GPU

The diagram shows the time in seconds for scube construction and total computation as a function of partial degree. The x-axis represents the partial degree, and the y-axis represents time in seconds. The graph indicates that as the partial degree increases, the time required for scube construction and total computation also increases.
Solving bivariate systems: timings

<table>
<thead>
<tr>
<th>$d$</th>
<th>$t_0$(gpu)</th>
<th>$t_1$(total)</th>
<th>$t_2$ (cpu)</th>
<th>$t_3$ (total)</th>
<th>$t_2/t_0$</th>
<th>$t_3/t_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>0.25</td>
<td>0.35</td>
<td>0.14</td>
<td>0.25</td>
<td>0.6</td>
<td>0.7</td>
</tr>
<tr>
<td>40</td>
<td>0.25</td>
<td>0.46</td>
<td>0.42</td>
<td>0.64</td>
<td>1.7</td>
<td>1.4</td>
</tr>
<tr>
<td>50</td>
<td>0.28</td>
<td>0.67</td>
<td>1.14</td>
<td>1.56</td>
<td>4.1</td>
<td>2.3</td>
</tr>
<tr>
<td>60</td>
<td>0.29</td>
<td>0.88</td>
<td>1.54</td>
<td>2.20</td>
<td>5.3</td>
<td>2.5</td>
</tr>
<tr>
<td>70</td>
<td>0.31</td>
<td>1.20</td>
<td>3.94</td>
<td>4.94</td>
<td>12.7</td>
<td>4.1</td>
</tr>
<tr>
<td>80</td>
<td>0.32</td>
<td>1.42</td>
<td>4.84</td>
<td>6.06</td>
<td>15.1</td>
<td>4.3</td>
</tr>
<tr>
<td>90</td>
<td>0.33</td>
<td>1.80</td>
<td>5.94</td>
<td>7.54</td>
<td>18.0</td>
<td>4.2</td>
</tr>
<tr>
<td>100</td>
<td>0.48</td>
<td>2.56</td>
<td>14.23</td>
<td>16.66</td>
<td>29.7</td>
<td>6.5</td>
</tr>
<tr>
<td>110</td>
<td>0.52</td>
<td>2.93</td>
<td>16.78</td>
<td>19.58</td>
<td>32.1</td>
<td>6.7</td>
</tr>
<tr>
<td>120</td>
<td>0.55</td>
<td>3.80</td>
<td>24.41</td>
<td>28.60</td>
<td>44.4</td>
<td>7.5</td>
</tr>
</tbody>
</table>

- $d$: total degree of the input polynomial
- $t_0$: GPU FFT based scube construction
- $t_1$: total time for solving with GPU code
- $t_2$: CPU FFT based scube construction
- $t_3$: total time for solving without GPU code
The Stockham FFT achieves a speedup factor of 21 for large FFT degrees, comparing to the \texttt{modpn} serial implementation.

The subresultant chain construction has been improved by a factor of (up to) 44 on the GPU.

For the bivariate solver, more code has to be ported to GPU (mainly univariate polynomial GCDs).

Nevertheless the GPU-based code solves within a second, polynomial systems for which pure serial code takes 7.5 sec.

The goal is to make bivariate and trivariate system solvers as fast as a univariate GCD routine in \texttt{Maple}.

Joint work with Wei Pan:
- Solving bivariate polynomial systems on a GPU. \textit{HPCS’2011}.
Plan

1. Hierarchical memories and cache complexity
2. Balanced polynomial arithmetic on multicores
3. Bivariate polynomial systems on the GPU
4. Status of our libraries
The RegularChains library in Maple

Specifications

- Solving polynomial systems with coefficients in $\mathbb{K}$ or $\mathbb{K}(t_1,\ldots,t_m)$ for $\mathbb{K} = \mathbb{Q}$ or $\mathbb{K} = \mathbb{F}_p$.
- Solves over $\overline{\mathbb{K}}$ with Triangularize and over $\mathbb{R}$ with RealTriangularize, SamplePoints, RealRootClassification, etc.
- Parametric system solving: ComprehensiveTriangularize and RealComprehensiveTriangularize.
- Operations on constructible sets and semi-algebraic sets: set-theoretic operations, projection, etc.

Features

- Use of types for algebraic structures: regular_chain, constructible_set, regular_semi_algebraic_system, semi_algebraic_set, etc.
- Growing support with C and CUDA libraries.
- > 100,000 lines of code and 140 UI commands.
C, Cilk++ and CDUA supporting libraries

**modpn (opaque module in MAPLE)**
- FTT-based dense multivariate arithmetic and SLPs
- Two UI’s: one in AXIOM and one in MAPLE: RegularChains:-FastArithmeticTools
- 40,000 lines of code, not documented.

**cumdp (in progress)**
- CUDA-based, so targeting GPUs
- Similar specification as modpn plus dense linear algebra.
- 20,000 lines of code, documented.
- Wei Pan, Anis Sardar Haque and Jiajiang Yang.

**BPAS (in progress)**
- Relies on modpn, cumdp and Spiral.
- Similar specification as modpn.
- Written in Cilk++, targeting multicores.
- Yuzhen Xie, Changbo Chen, Mohsin Ali, Zunaid Haque.