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^α coefficient operations, up to a small constant say 2 to 10.
- A typical naive implementation will incur n^α/L cache misses, which reduce to $n^\alpha/(\sqrt{Z}L)$ for a cache-friendly algorithm.
- Moreover, execution and memory models (say multicore vs manycore) have an impact on algorithm design.

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.

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.

Et le calcul formel dans tout cela ?

- Typical algorithms have high algebraic complexity, say n^α 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.

Et le calcul formel dans tout cela ?

- Typical algorithms have high algebraic complexity, say n^α 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 $\mathbb{Z}$.

Et le calcul formel dans tout cela ?

- Typical algorithms have high algebraic complexity, say n^α 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 $\mathbb{Z}$.
- As mentioned before, the algebraic-complexity-to-cache-complexity ratio is often a constant: bad!

Et le calcul formel dans tout cela ?

- Typical algorithms have high algebraic complexity, say n^α 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 $\{\backslash$*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.

Et le calcul formel dans tout cela ?

- Typical algorithms have high algebraic complexity, say n^α 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 $\{\backslash$*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.

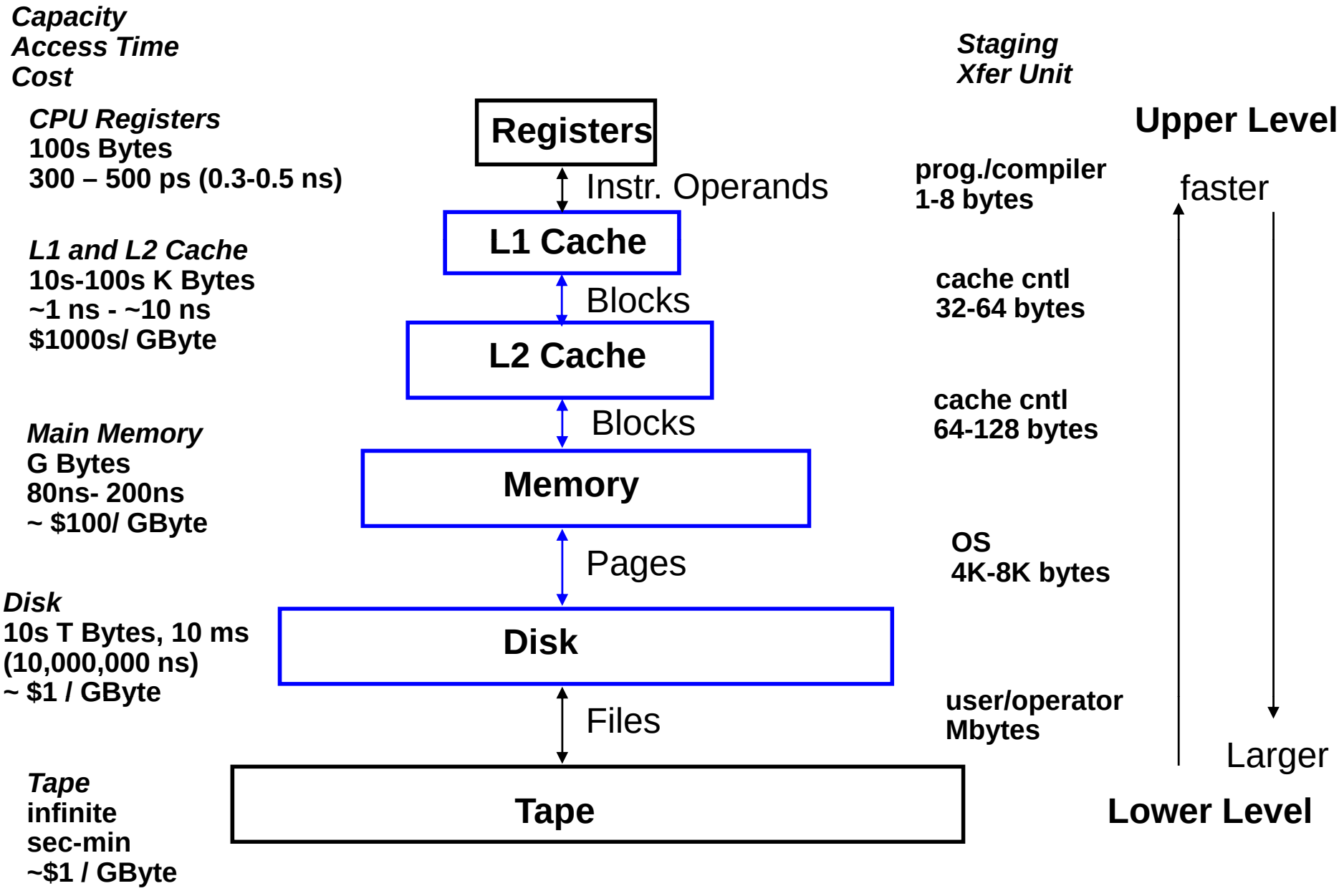
Et le calcul formel dans tout cela ?

- Typical algorithms have high algebraic complexity, say n^α 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 $\mathbb{Z}$.
- 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!

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

Plan


Capacity
Access Time
Cost
CPU Registers
100s Bytes
300 – 500 ps (0.3-0.5 ns)
L1 and L2 Cache
10s-100s K Bytes
~1 ns - ~10 ns
$1000s/ GByte
Main Memory
G Bytes
80ns- 200ns
~ $100/ GByte
Disk
10s T Bytes, 10 ms
(10,000,000 ns)
~ $1 / GByte
Tape
infinite
sec-min
~$1 / GByte
Registers
Instr. Operands
L1 Cache
Blocks
L2 Cache
Blocks
Memory
Pages
Disk
Files
Tape
Staging
Xfer Unit
prog./compiler
1-8 bytes
cache cntl
32-64 bytes
cache cntl
64-128 bytes
OS
4K-8K bytes
user/operator
Mbytes
Upper Level
faster
Larger
Lower Level

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

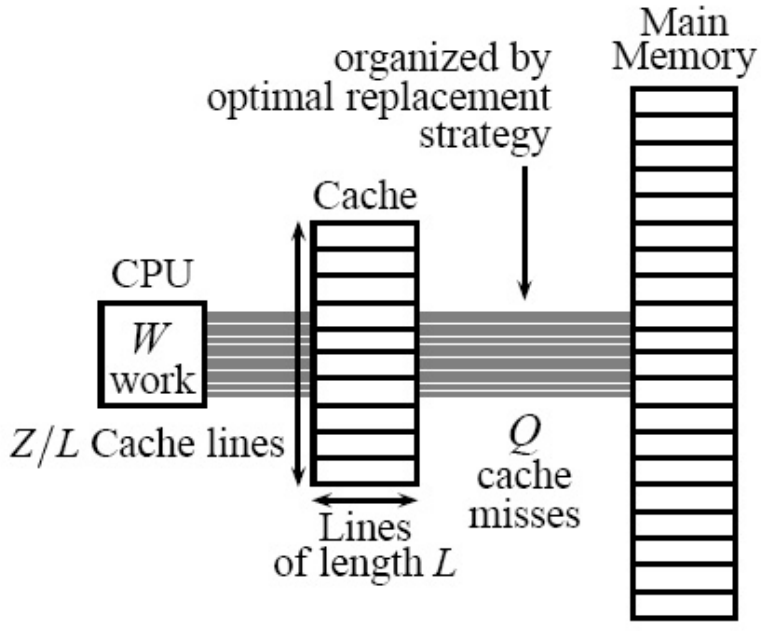


Figure 1: The ideal-cache model

- 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.

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

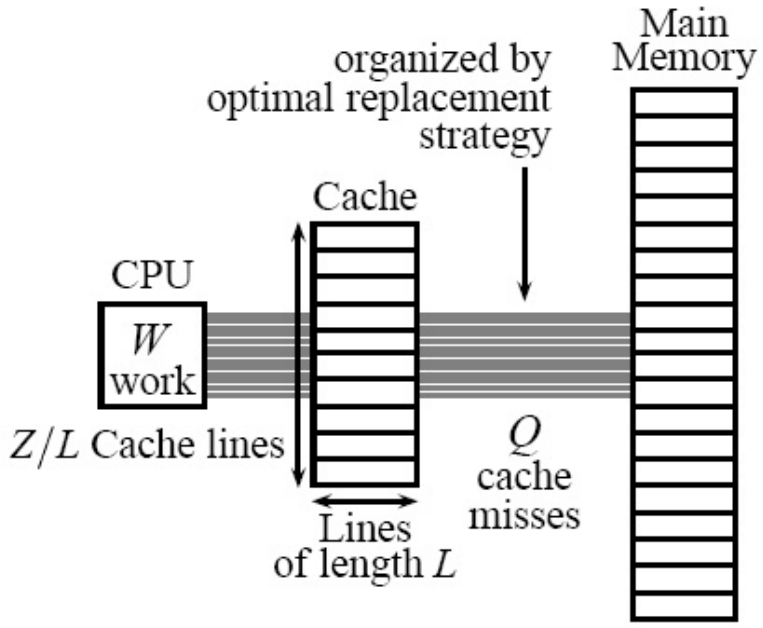


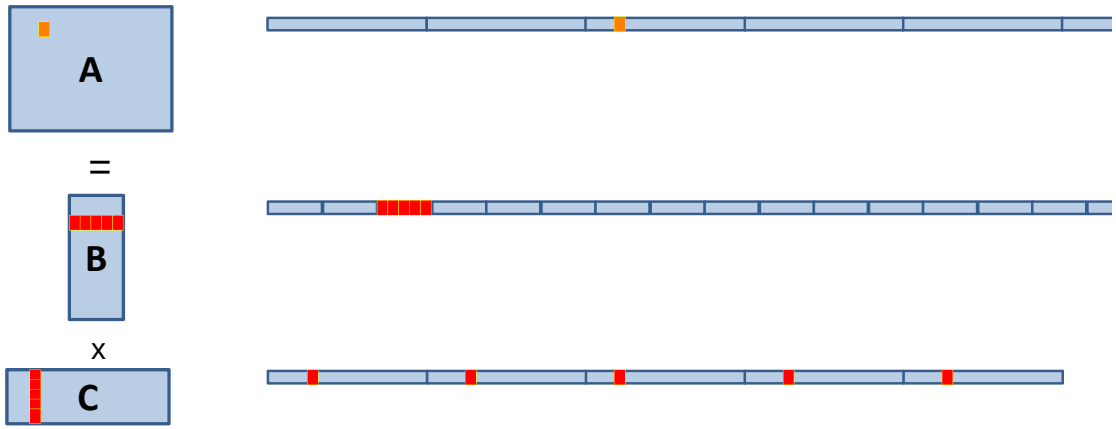
Figure 1: The ideal-cache model

- 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.

A typical naive matrix multiplication C code

```
#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

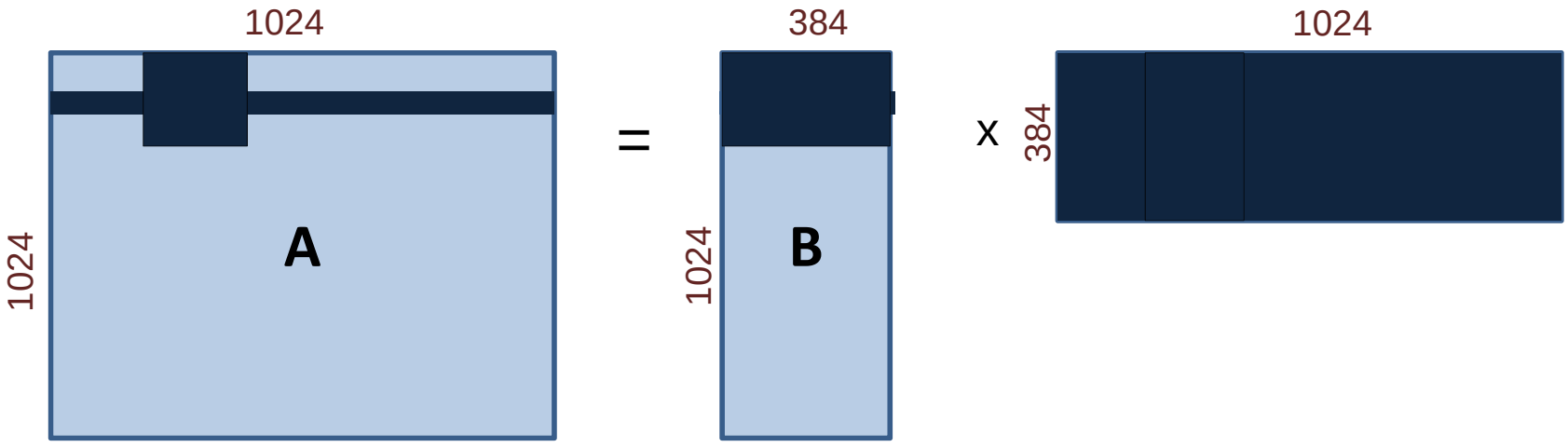


- Let A, B and C have format (m, n), (m, p) and (p, n) respectively.
- A is scanned one, so mn/L cache misses if L is the number of coefficients per cache line.
- B is scanned n times, so mnp/L cache misses if the cache cannot hold a row.
- C is accessed "nearly randomly" (for m large enough) leading 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

```
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;
}
```

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 $O(n^3/(\sqrt{Z}L))$ cache misses, if B is well chosen, which is optimal.

Transposing and blocking for optimizing data locality

```
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)

n	naive	transposed	speedup	64×64-tiled	speedup	t. & t.	speedup
128	7	3		7		2	
256	26	43		155		23	
512	1805	265	6.81	1928	0.936	187	9.65
1024	24723	3730	6.62	14020	1.76	1490	16.59
2048	271446	29767	9.11	112298	2.41	11960	22.69
4096	2344594	238453	**9.83**	1009445	**2.32**	101264	**23.15**

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).

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.

	CPI		L1 Miss Rate		L2 Miss Rate	Percent SSE Instructions	Instructions Retired	
In C	4.78		0.24		0.02	43%	13,137,280,000	
		5x		2x				1x
Transposed	1.13		0.15		0.02	50%	13,001,486,336	
		3x		8x				0.8x
Tiled	0.49		0.02		0	39%	18,044,811,264	

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 Rec-Transpose below incurs $\Theta(1 + mn/L)$ cache misses, which is optimal.
  1. If $n \geq m$, the Rec-Transpose algorithm partitions

$$A = (A_1 \ A_2) \ , \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

     and recursively executes Rec-Transpose(A_1, B_1) and Rec-Transpose(A_2, B_2).
  2. If $m > n$, the Rec-Transpose algorithm partitions

$$A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \ , \quad B = (B_1 \ B_2)$$

     and recursively executes Rec-Transpose(A_1, B_1) and Rec-Transpose(A_2, B_2).

Cache-oblivious matrix transposition into practice

```
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!

size	Naive	Cache-oblivious	ratio
5000x5000	126	79	1.59
10000x10000	627	311	2.02
20000x20000	4373	1244	3.52
30000x30000	23603	2734	8.63
40000x40000	62432	4963	12.58

- `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:

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} , \tag{1}$$

$$\begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 , \tag{2}$$

$$A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} . \tag{3}$$

- 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.

Summary and notes

- 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

FFTs over finite fields on multicores

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.

FFT-based multivariate multiplication

- 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 $s_i \geq d_i + d'_i + 1$.

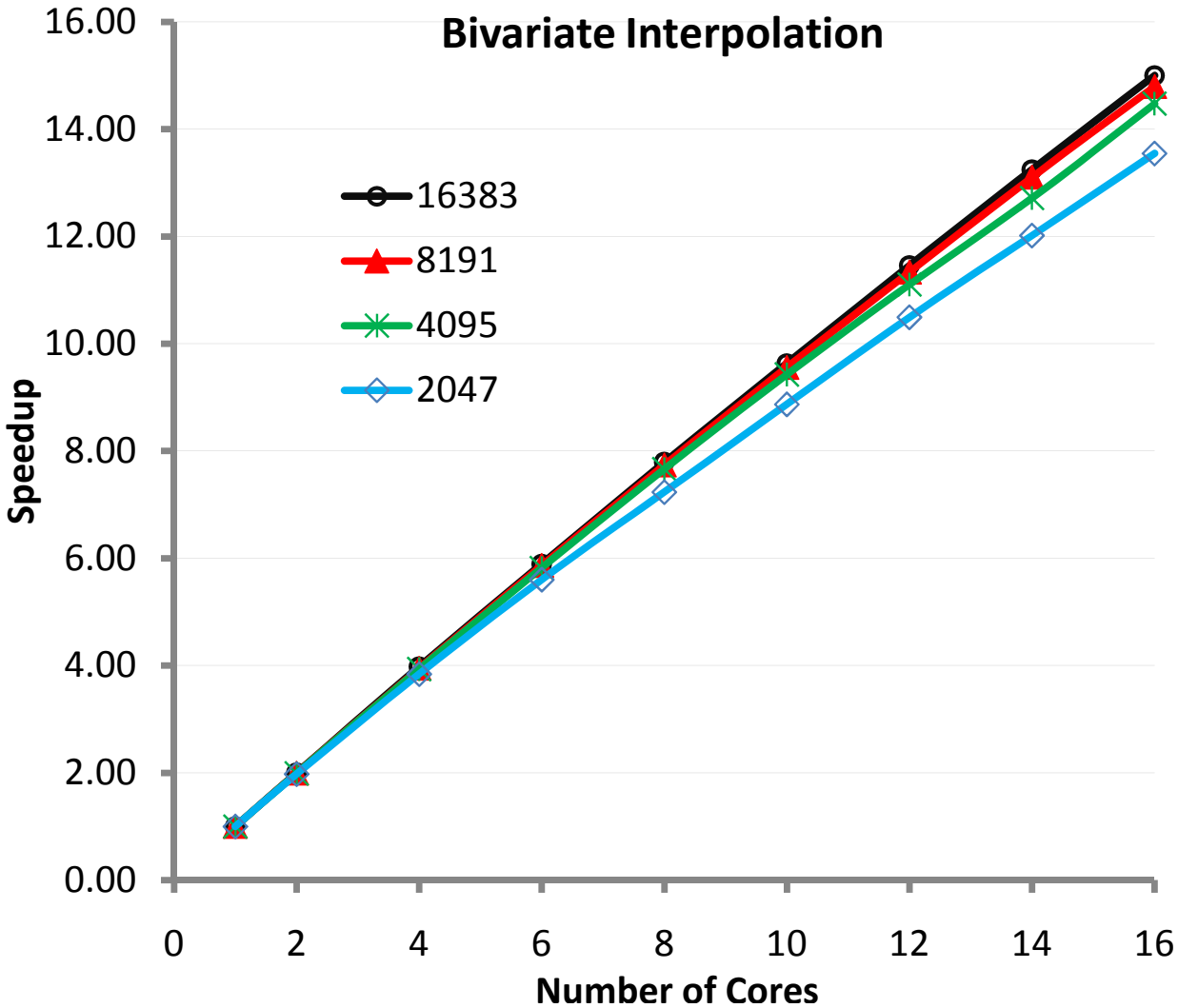
Then fg can be computed as follows.

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 $f(P)g(P)$,

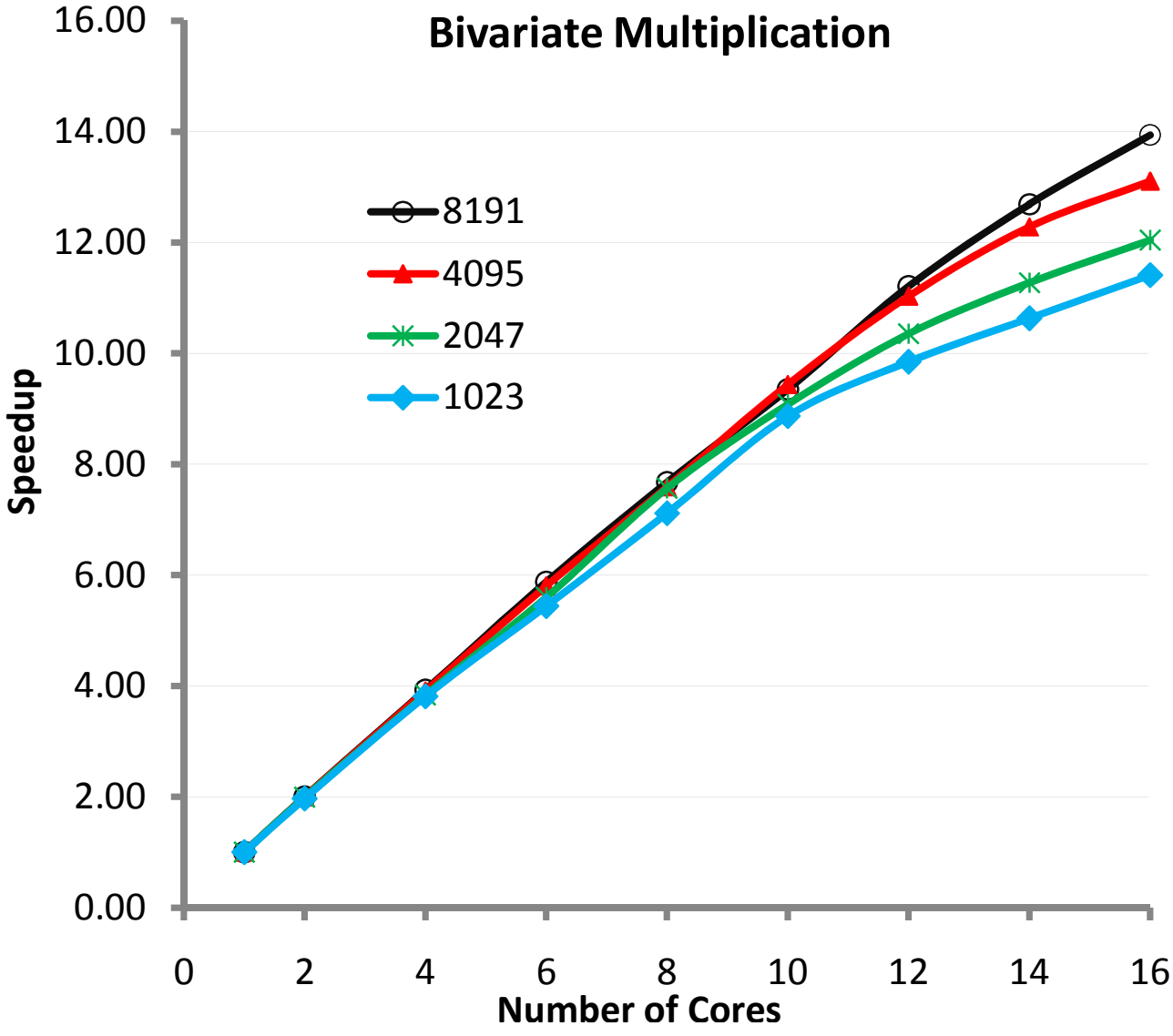
Step 3. Interpolate fg (from its values on the grid) via n-D FFT.

Speedup factors of bivariate interpolation $(d_1 = d_2)$

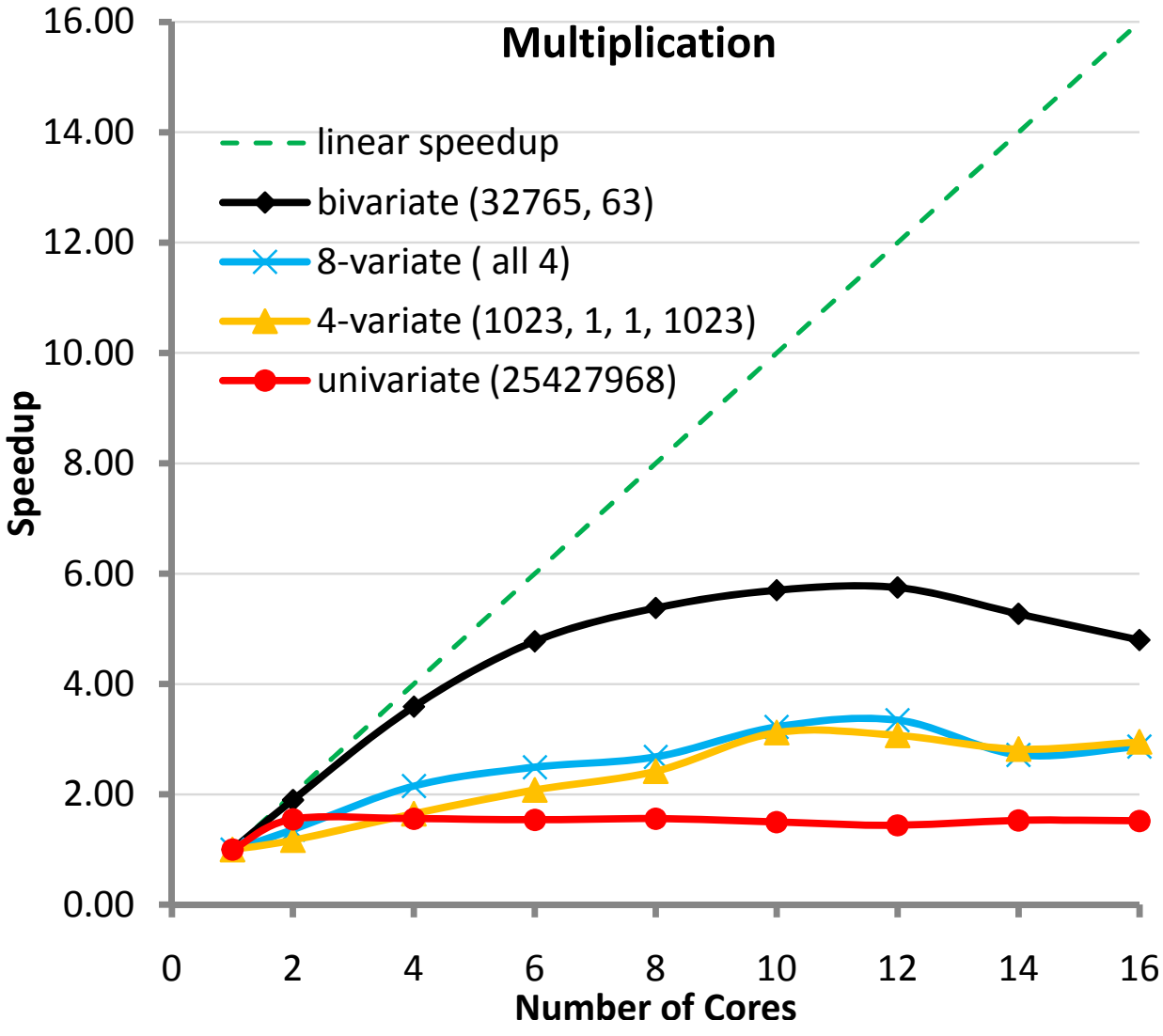


Special thanks to Matteo Frigo for his cache-efficient code for matrix transposition!

Speedup factors of bivariate multiplication $(d_1 = d_2 = d'_1 = d'_2)$



Challenges: irregular input data



Performance analysis with VTune

No.	Size of Two Input Polynomials	Product Size
1	8191×8191	268402689
2	259575×258	268401067
3	63×63×63×63	260144641
4	8 vars. of deg. 5	214358881

No.	INST_ RETIRED. ANY×10^9	Clocks per Instruction Retired	L2 Cache Miss Rate ($\times 10^{-3}$)	Modified Data Sharing Ratio ($\times 10^{-3}$)	Time on 8 Cores (s)
1	659.555	0.810	0.333	0.078	16.15
2	713.882	0.890	0.735	0.192	19.52
3	714.153	0.854	1.096	0.635	22.44
4	1331.340	1.418	1.177	0.576	72.99

Complexity analysis (1/2)

- 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 *Step* 1 is

$$\tfrac{9}{2}\left(s_1 \lg(s_1) + \cdots + s_n \lg(s_n)\right),$$

and the parallelism of *Step* 1 is lower bounded by

$$s/\max(s_1, \ldots, s_n). \tag{4}$$

- Let L be the size of a cache line. For some constant $c > 0$, the number of cache misses of *Step* 1 is upper bounded by

$$n\frac{cs}{L} + cs(\frac{1}{s_1} + \cdots + \frac{1}{s_n}). \tag{5}$$

Complexity analysis (2/2)

- Let $Q(s_1, \ldots, s_n)$ denotes the total number of cache misses for the whole algorithm, for some constant c we obtain

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

- 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

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

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 $s = s_1 s_2$, 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

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

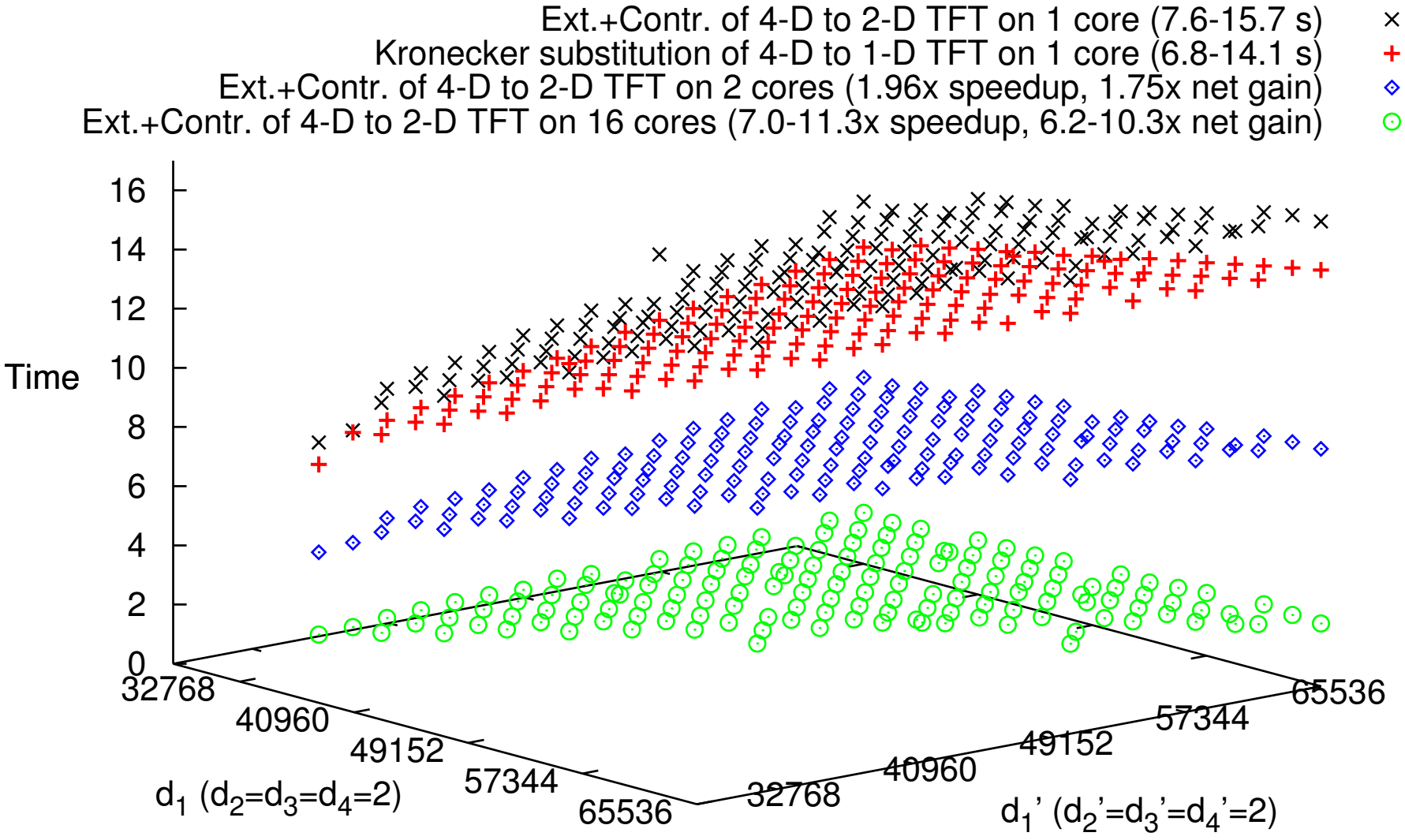
Step 2. Direct Kronecker substitution $\{v, x_2, \ldots, x_n\}$ to v.

such that

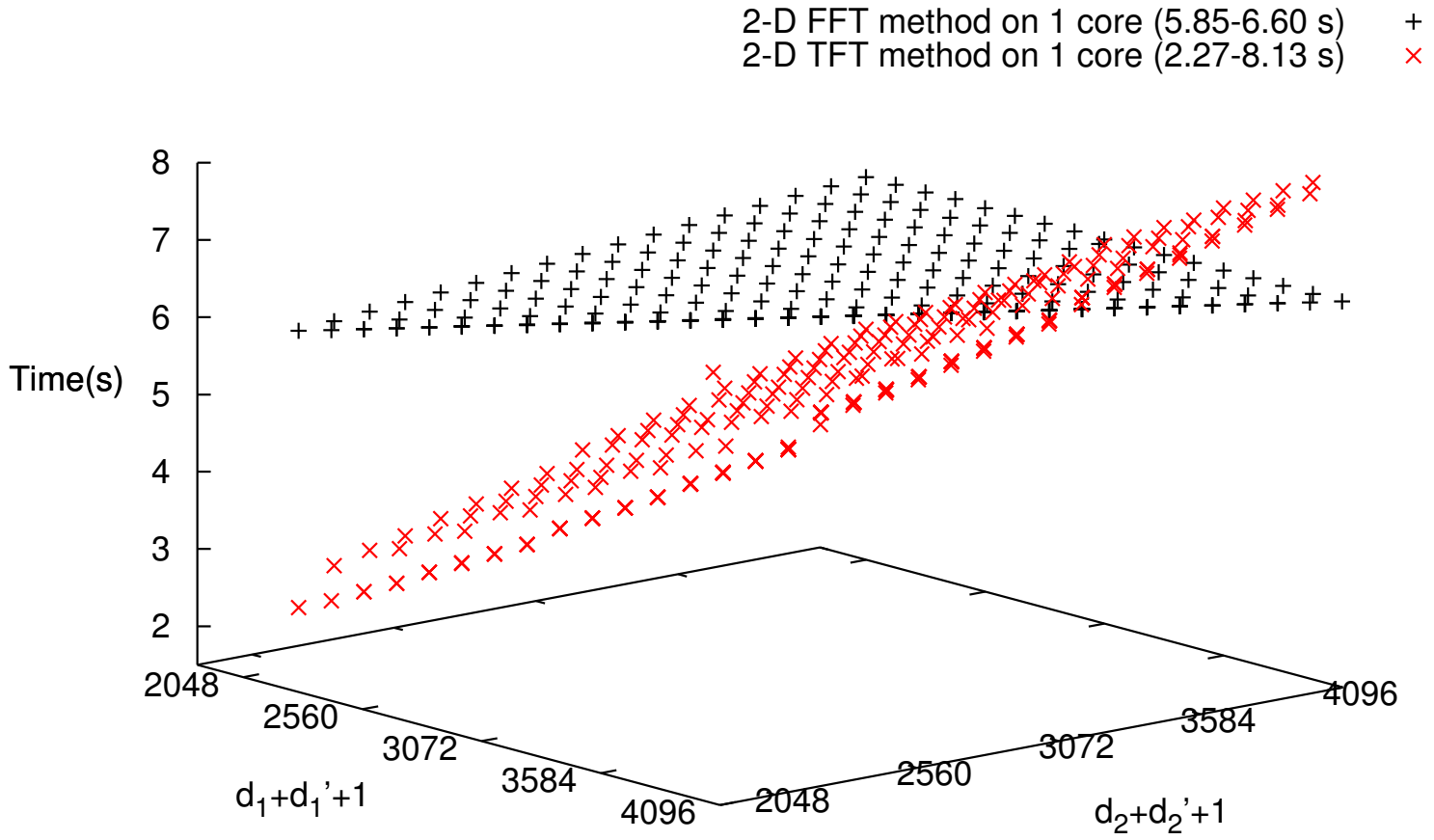
- the pair f_b, g_b is (nearly) a balanced pair and $f_b g_b$ has dense size at most twice that of fg.
- we can recover the product fg from the product $f_b g_b$

M^3, Yuzhen Xie: Balanced Dense Polynomial Multiplication on Multi-Cores. *Int. J. Found. Comput. Sci.*, 2011.

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

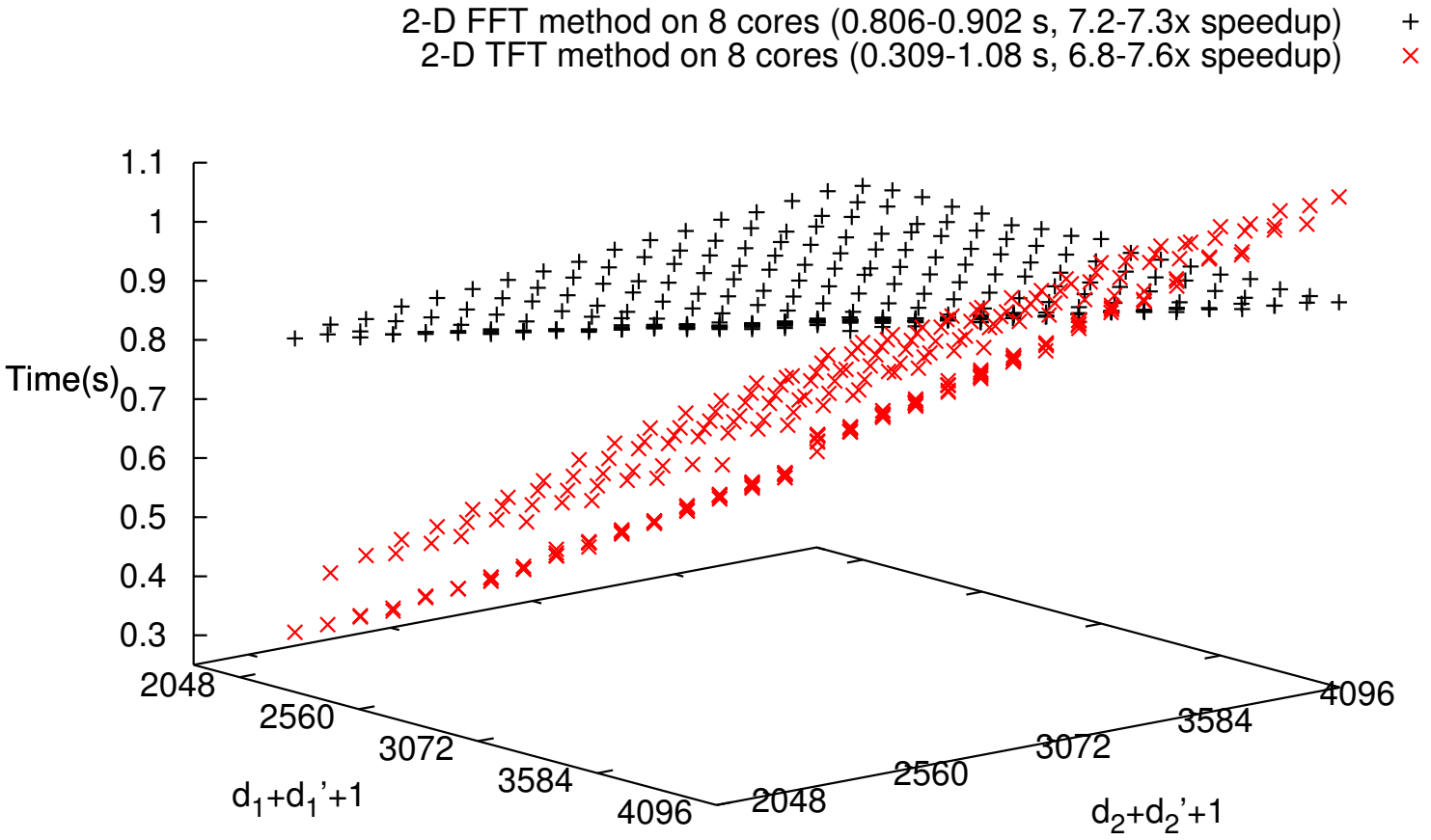


Quizz for Joris (1/3)



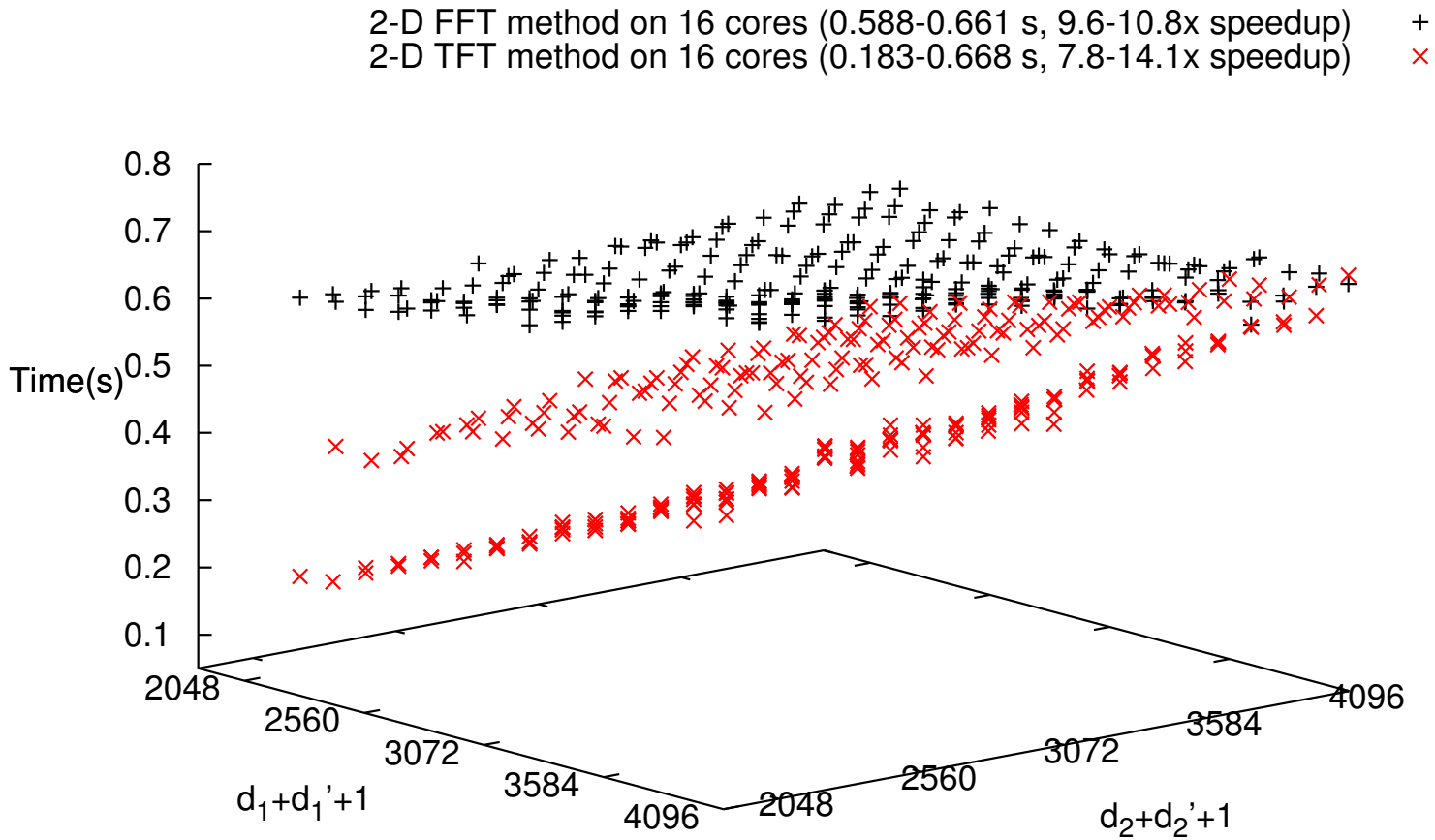
Bivariate multiplication for input degree range of $[1024, 2048)$ on 1 core.

Quizz for Joris (2/3)



Bivariate multiplication for input degree range of $[1024, 2048)$ on 8 cores.

Quizz for Joris (3/3)



Bivariate multiplication for input degree range of $[1024, 2048)$ on 16 cores.

Question: why TFT always beats FFT on 16 cores?

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!
- M^3, Yuzhen Xie: FFT-based Dense Polynomial Arithmetic on Multi-cores in *Proc. HPCS'2009*.

Plan

Background

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

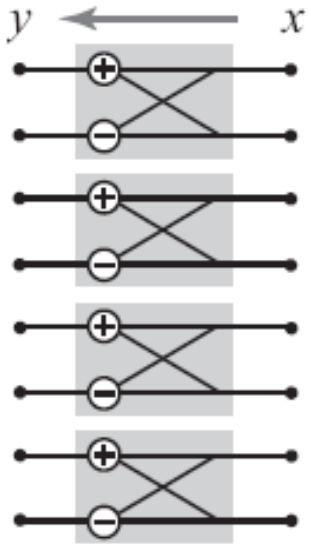
$\log_2 n$	memset	Main Mem to GPU	GPU to Main Mem	GPU Kernel
23	1.56	1.33	1.52	61.6
24	1.56	1.34	1.52	69.9
25	1.39	1.35	1.53	75.0
26	1.39	1.28	1.50	77.4
27	1.43	1.35	1.49	79.0

- Intel Core 2 Quad Q9400 @ 2.66GHz, 6GB memory, memory interface width 128 bits
- GeForce GTX 285, 1GB global memory, 30×8 cores, memory interface

Extract parallelism from structural formulas

$I_n \otimes A$: block parallelism

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

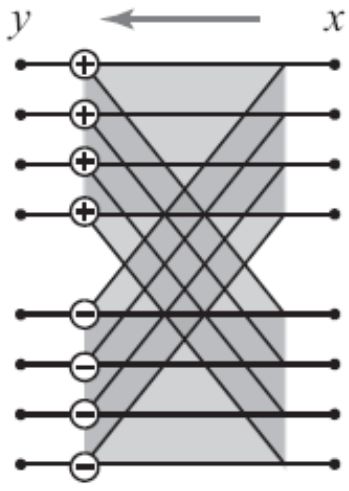


How To Write Fast Numerical Code: A Small Introduction by Srinivas Chellappa, Franz Franchetti, and Markus Pueschel.

Extract parallelism from structural formulas

$A \otimes I_n$: vector parallelism

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



Stockham FFT

$$\mathrm{DFT}_{2^k} = \prod_{i=0}^{k-1} \underbrace{(\mathrm{DFT}_2 \otimes I_{2^{k-1}})}_{\text{butterfly}} \underbrace{(D_{2,2^{k-i-1}} \otimes I_{2^i})}_{\text{twiddling}} \underbrace{(L_2^{2^{k-i}} \otimes I_{2^i})}_{\text{reordering}}$$

```
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

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

with the twiddle factor matrix $T_{n,i} = I_{2^{i-1}} \otimes D_{2,2^{k-i}}$ 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

e	modpn	Cooley-Tukey		C-T + Mem		Stockham		S + Mem	
		time	ratio	time	ratio	time	ratio	time	ratio
12	1	1	1.0	1	1.0	2	0.5	2	0.5
13	1	2	0.5	2	0.5	2	0.5	3	0.3
14	3	1	3.0	2	1.5	2	1.5	3	1.0
15	4	2	2.0	2	2.0	3	2.0	3	1.3
16	10	3	3.3	3	3.3	3	3.3	4	3.3
17	16	4	4.0	5	3.2	3	5.3	5	3.2
18	37	6	6.2	9	4.1	4	9.3	7	5.3
19	71	11	6.5	15	6.5	6	11.8	10	7.1
20	174	22	7.9	28	6.2	9	19.3	16	10.9
21	470	44	10.7	56	8.4	16	29.4	28	16.8
22	997	83	12.0	105	9.5	29	34.4	52	19.2
23	2070	165	12.5	210	9.9	56	37.0	101	20.5
24	4194	330	12.7	418	10.0	113	37.0	201	20.9
25	8611	667	12.9	842	10.2	230	37.4	405	21.2
26	17617	1338	13.2	1686	10.4	473	37.2	822	21.4

The GPU is GTX 285.

Solving polynomial systems with GPU support

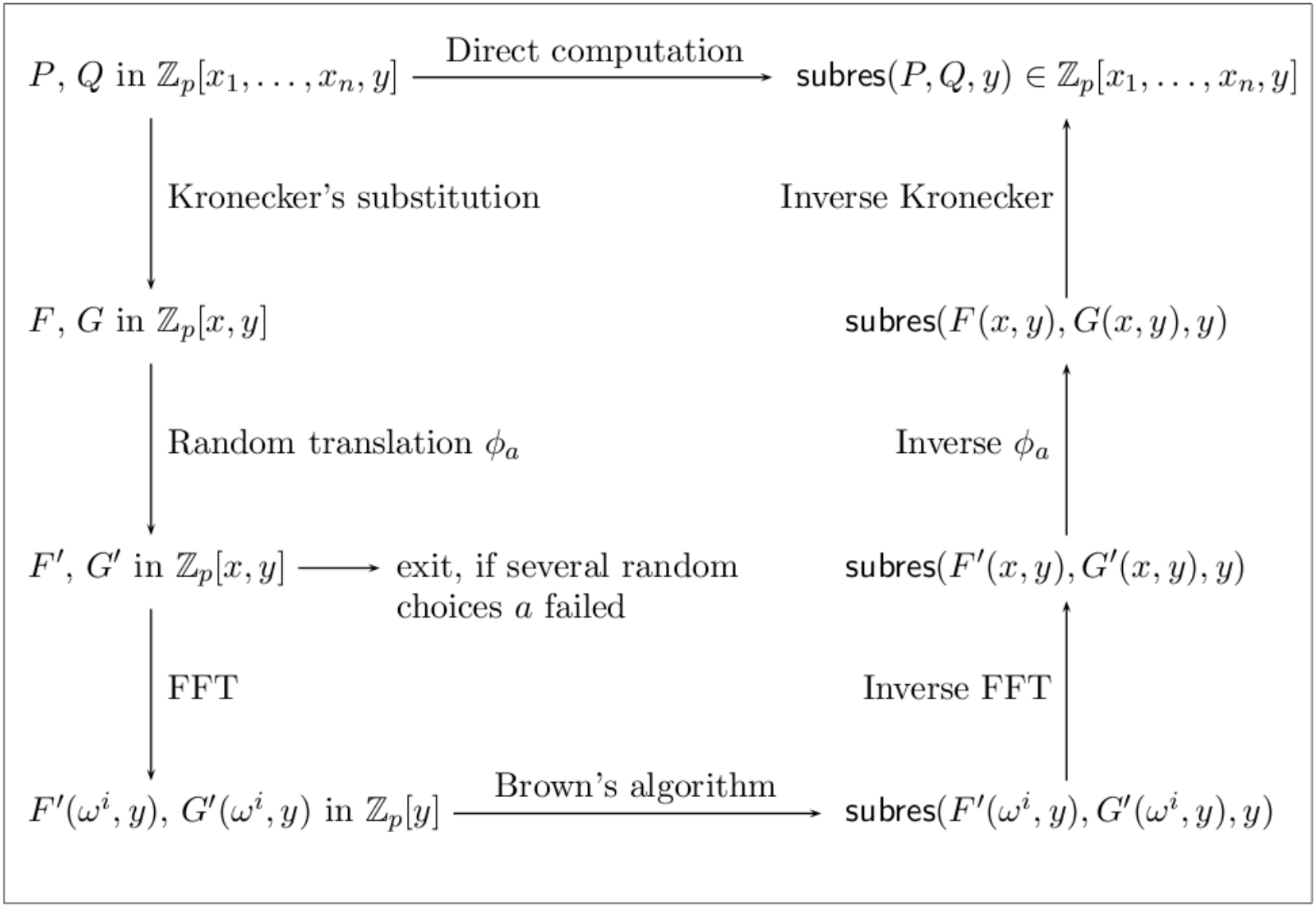
Main idea

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



Subresultant chain by evaluation/interpolation

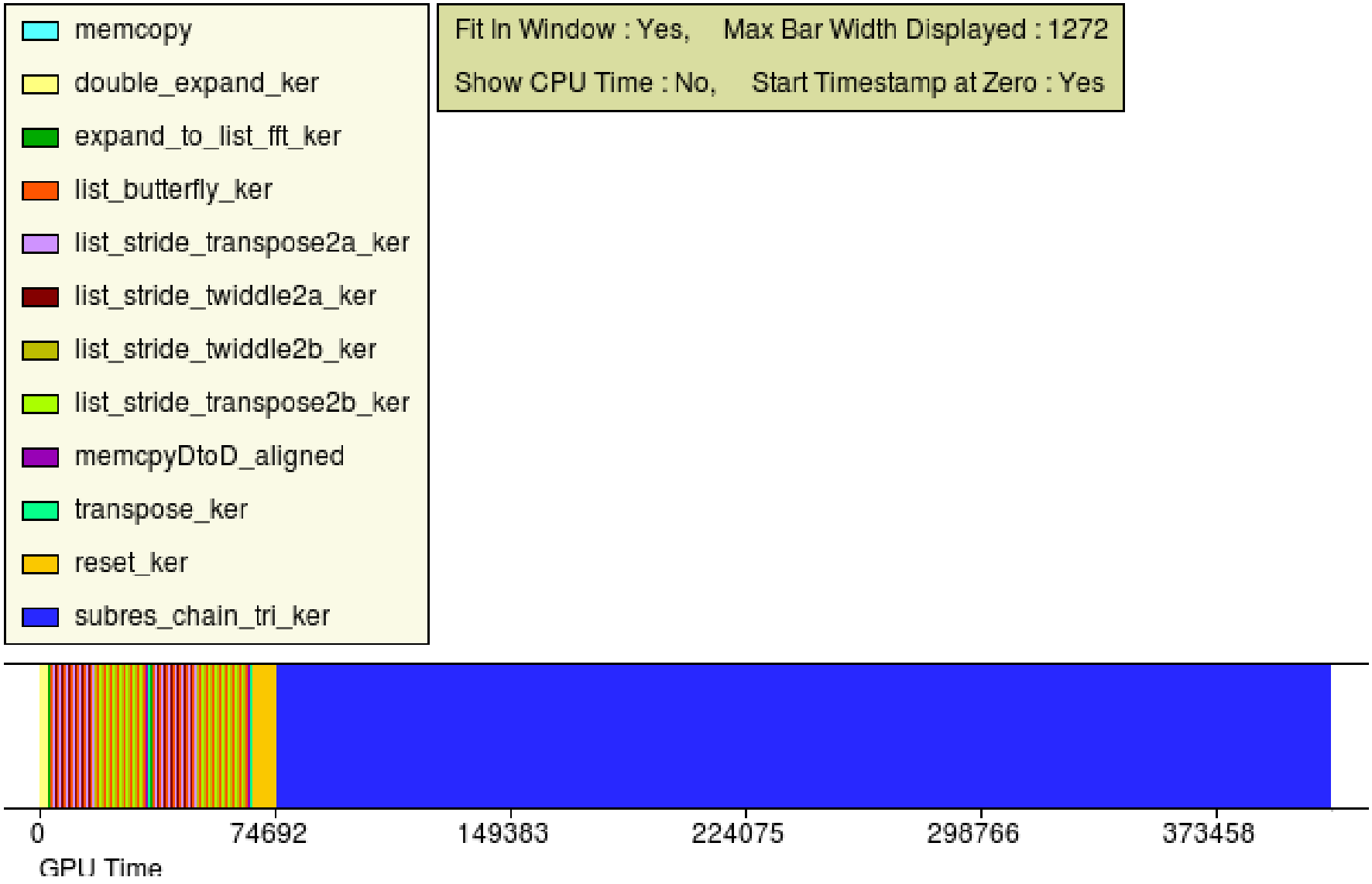
Issues with different strategies

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

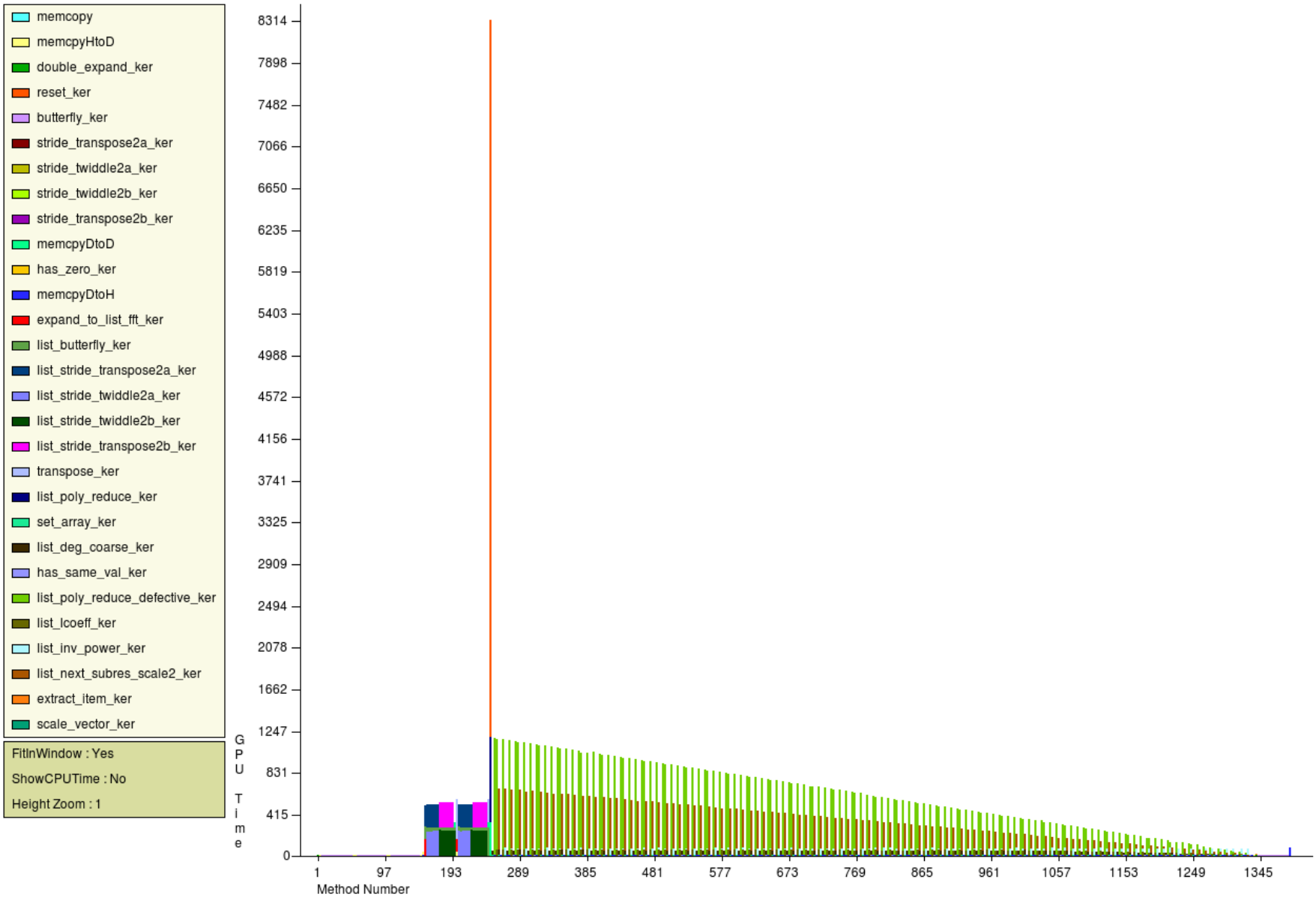
FFT scube on the GPU. Two approaches:

- 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



Profiling fine-grained implementation



Computing resultants

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

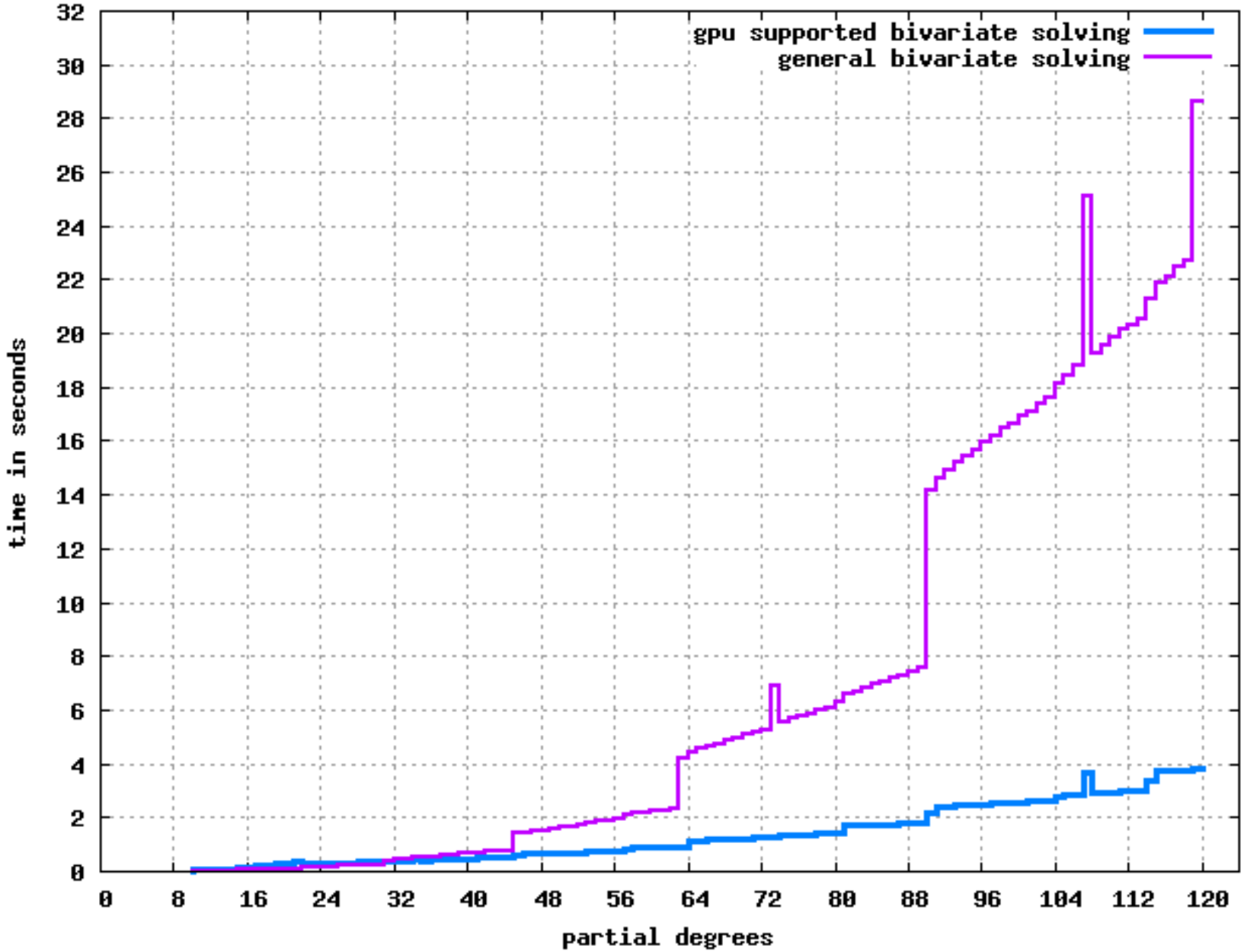
Bivariate dense polynomials of total degree d.

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

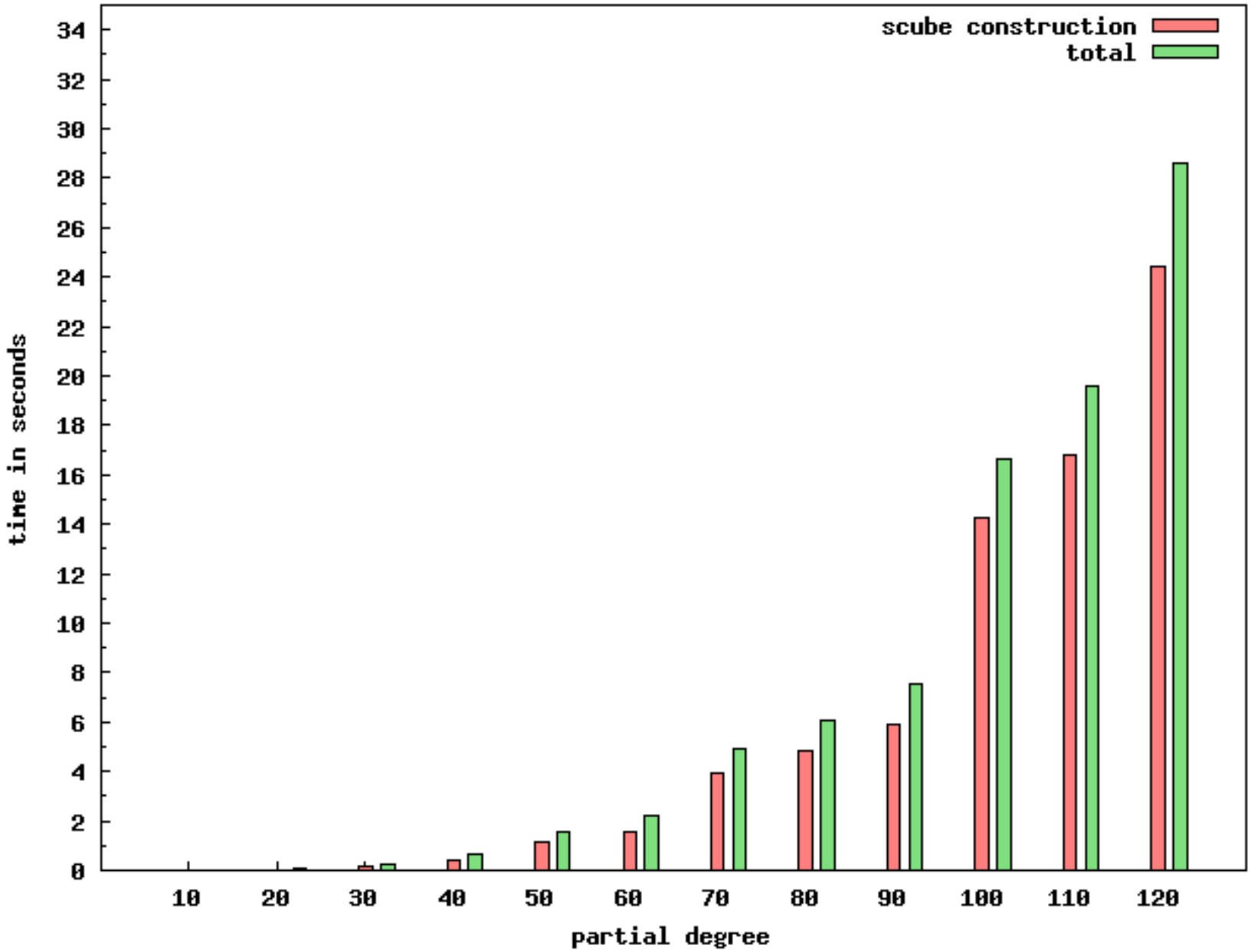
Trivariate dense polynomials of total degree d.

- t_0, GPU fft code
- t_1, CPU fft code
- Nvidia Tesla C2050

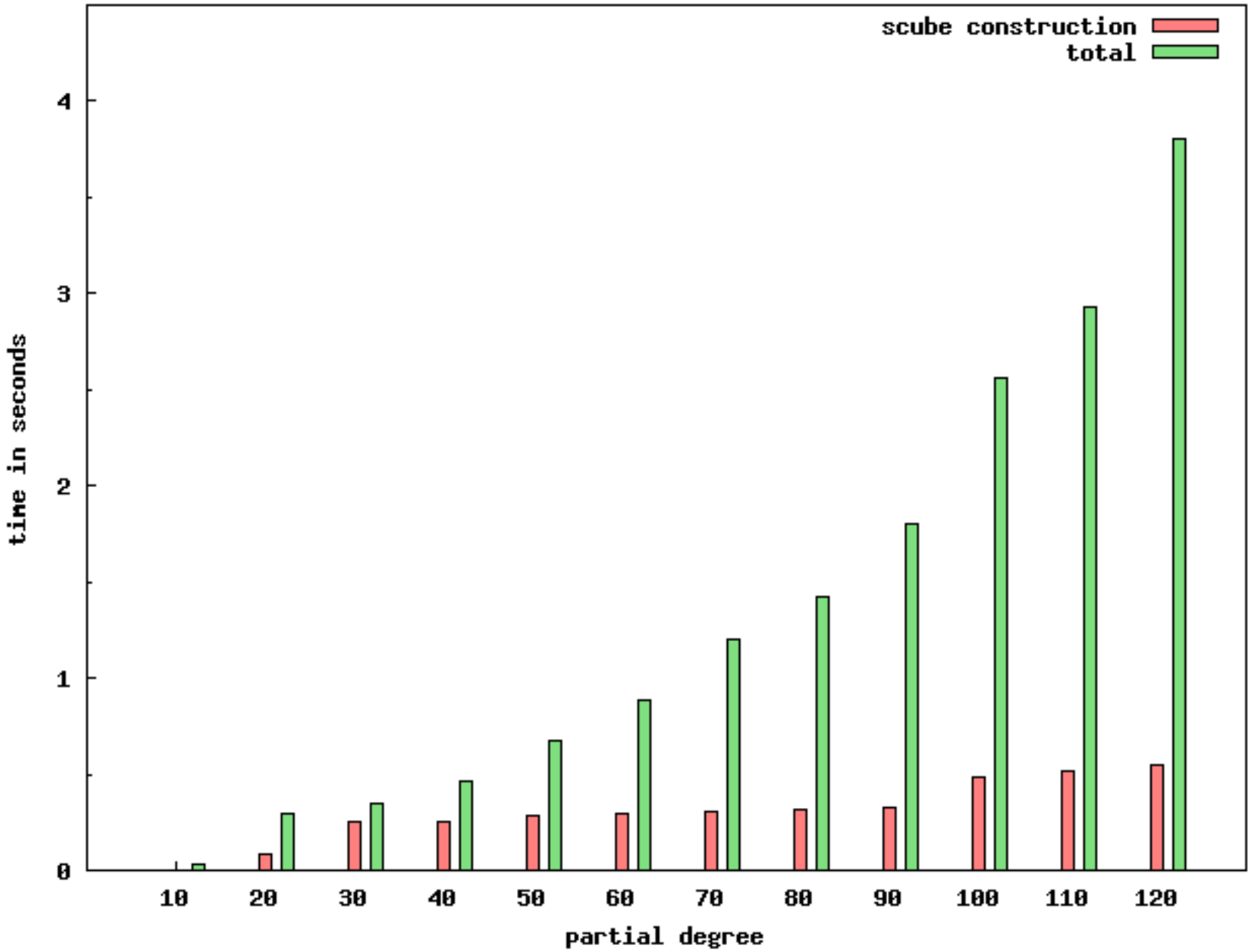
Bivariate solver



Bivariate solver on the CPU



Bivariate solver on the GPU



Solving bivariate systems: timings

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

- 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

Summary and notes

- The Stockham FFT achieves a speedup factor of 21 for large FFT degrees, comparing to the `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 MAPLE.
- Joint work with Wei Pan:
  - Fast polynomial multiplication on a GPU. *Journal of Physics, Conference Series*, 2010.
  - Solving bivariate polynomial systems on a GPU. *HPCS'2011*.

Plan

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.