# Solving Bivariate Polynomial Systems on the GPU

Marc Moreno Maza and Wei Pan

University of Western Ontario and Intel Corporation

AMMCS, Waterloo, 25 July 2011

# Plan

# Plan

# Background

## Motivation

- Solving bivariate and trivariate polynomial systems is a kernel in many algorithms involving polynomials.
- Actually some algorithms solving arbitrary polynomial systems essentially reduce to these two cases.
- The horse power of GPUs offer the opportunity for symbolic (and thus exact) computation to provide efficient implementation of that kernel even for large input problems (say with $10,000$ solutions).

## The bivariate case

- Very attractive: it is well understood and probably has nearly optimal algorithmic solutions.
- Moreover, its 2-D nature makes it well suited for the data-parallelism of the GPU programming model.

# Background

## Motivation

- Solving bivariate and trivariate polynomial systems is a kernel in many algorithms involving polynomials.
- Actually some algorithms solving arbitrary polynomial systems essentially reduce to these two cases.
- The horse power of GPUs offer the opportunity for symbolic (and thus exact) computation to provide efficient implementation of that kernel even for large input problems (say with $10,000$ solutions).

## The bivariate case

- Very attractive: it is well understood and probably has nearly optimal algorithmic solutions.
- Moreover, its 2-D nature makes it well suited for the data-parallelism of the GPU programming model.

# A bivariate system example

- Consider

$$P = \left(y^2+6\right)(x-1) - y\left(x^2+1\right), Q = \left(x^2+6\right)(y-1) - x\left(y^2+1\right)$$

- To solve $P = Q = 0$, we can first "eliminate" $y$, running an **Eulcidean-like Algorithm** leading to their **resultant w.r.t. y**:

$$\mathrm{res}(P, Q, y) = 2\left(x^2 - x + 4\right)(x-2)^2(x-3)^2.$$

- Then, for each root $x = x_0$, we compute $\gcd(P(x_0, y), Q(x_0, y))$:
  1. $\mathrm{gcd}(\mathrm{P}, \mathrm{Q}, \mathrm{x} = 2) = (\mathrm{y} - 2)(\mathrm{y} - 3)$.
  2. $\mathrm{gcd}(\mathrm{P}, \mathrm{Q}, \mathrm{x} = 3) = (\mathrm{y} - 2)(\mathrm{y} - 3)$.
  3. $\mathrm{gcd}(\mathrm{P}, \mathrm{Q}, \mathrm{x}^2 - \mathrm{x} + 4 = 0) = (2\,\mathrm{x} - 1)\,\mathrm{y} - 7 - \mathrm{x}$.
- One major trouble occurs: for roots defined symbolically such as those of $x^2 - x + 4$, we are repeating (modulo $x^2 - x + 4 = 0$) the computations performed to get $\mathrm{res}(P, Q, y)$.
- Solution: we store all the computations performed to get $\mathrm{res}(P, Q, y)$ and use the algorithm of (Xin Li, Moreno Maza & Wei Pan, ISSAC 2009) to deduce (at essentially no costs) the GCDs.

# A bivariate system example

- Consider

$$P = \left(y^2+6\right)(x-1) - y\left(x^2+1\right), Q = \left(x^2+6\right)(y-1) - x\left(y^2+1\right)$$

- To solve $P = Q = 0$, we can first "eliminate" $y$, running an **Eulcidean-like Algorithm** leading to their **resultant w.r.t. y**:

$$\mathrm{res}(P, Q, y) = 2\left(x^2 - x + 4\right)(x-2)^2(x-3)^2.$$

- Then, for each root $x = x_0$, we compute $\gcd(P(x_0, y), Q(x_0, y))$:
  1. $\gcd(\mathrm{P}, \mathrm{Q}, \mathrm{x} = 2) = (\mathrm{y}-2)(\mathrm{y}-3)$.
  2. $\gcd(\mathrm{P}, \mathrm{Q}, \mathrm{x} = 3) = (\mathrm{y}-2)(\mathrm{y}-3)$.
  3. $\gcd(\mathrm{P}, \mathrm{Q}, \mathrm{x}^2 - \mathrm{x} + 4 = 0) = (2\,\mathrm{x}-1)\,\mathrm{y} - 7 - \mathrm{x}$.
- One major trouble occurs: for roots defined symbolically such as those of $x^2 - x + 4$, we are repeating (modulo $x^2 - x + 4 = 0$) the computations performed to get $\mathrm{res}(P, Q, y)$.
- Solution: we store all the computations performed to get $\mathrm{res}(P, Q, y)$ and use the algorithm of (Xin Li, Moreno Maza & Wei Pan, ISSAC 2009) to deduce (at essentially no costs) the GCDs.

# A bivariate system example

- Consider

$$P = \left(y^2+6\right)(x-1) - y\left(x^2+1\right), Q = \left(x^2+6\right)(y-1) - x\left(y^2+1\right)$$

- To solve $P = Q = 0$, we can first "eliminate" $y$, running an **Eulcidean-like Algorithm** leading to their **resultant w.r.t. y**:

$$\mathrm{res}(P, Q, y) = 2\left(x^2 - x + 4\right)(x-2)^2(x-3)^2.$$

- Then, for each root $x = x_0$, we compute $\gcd(P(x_0, y), Q(x_0, y))$:
  1. $\mathrm{gcd}(\mathrm{P}, \mathrm{Q}, \mathrm{x} = 2) = (\mathrm{y} - 2)(\mathrm{y} - 3)$.
  2. $\mathrm{gcd}(\mathrm{P}, \mathrm{Q}, \mathrm{x} = 3) = (\mathrm{y} - 2)(\mathrm{y} - 3)$.
  3. $\mathrm{gcd}(\mathrm{P}, \mathrm{Q}, \mathrm{x}^2 - \mathrm{x} + 4 = 0) = (2\,\mathrm{x} - 1)\,\mathrm{y} - 7 - \mathrm{x}$.
- One major trouble occurs: for roots defined symbolically such as those of $x^2 - x + 4$, we are repeating (modulo $x^2 - x + 4 = 0$) the computations performed to get $\mathrm{res}(P, Q, y)$.
- Solution: we store all the computations performed to get $\mathrm{res}(P, Q, y)$ and use the algorithm of (Xin Li, Moreno Maza & Wei Pan, ISSAC 2009) to deduce (at essentially no costs) the GCDs.

# A bivariate system example

- Consider

$$P = \left(y^2 + 6\right)(x-1) - y\left(x^2+1\right), Q = \left(x^2+6\right)(y-1) - x\left(y^2+1\right)$$

- To solve $P = Q = 0$, we can first "eliminate" $y$, running an **Eulcidean-like Algorithm** leading to their **resultant w.r.t. y**:

$$\mathrm{res}(P, Q, y) = 2\left(x^2 - x + 4\right)(x-2)^2(x-3)^2.$$

- Then, for each root $x = x_0$, we compute $\gcd(P(x_0, y), Q(x_0, y))$:
  1. $\mathrm{gcd(P, Q, x = 2) = (y - 2)(y - 3)}$.
  2. $\mathrm{gcd(P, Q, x = 3) = (y - 2)(y - 3)}$.
  3. $\mathrm{gcd(P, Q, x^2 - x + 4 = 0) = (2\,x - 1)\,y - 7 - x}$.
- One major trouble occurs: for roots defined symbolically such as those of $x^2 - x + 4$, we are repeating (modulo $x^2 - x + 4 = 0$) the computations performed to get $\mathrm{res}(P, Q, y)$.
- Solution: we store all the computations performed to get $\mathrm{res}(P, Q, y)$ and use the algorithm of (Xin Li, Moreno Maza & Wei Pan, ISSAC 2009) to deduce (at essentially no costs) the GCDs.

# A bivariate system example

- Consider

$$P = \left(y^2+6\right)(x-1) - y\left(x^2+1\right), Q = \left(x^2+6\right)(y-1) - x\left(y^2+1\right)$$

- To solve $P = Q = 0$, we can first "eliminate" $y$, running an **Eulcidean-like Algorithm** leading to their **resultant w.r.t. y**:

$$\mathrm{res}(P, Q, y) = 2\left(x^2 - x + 4\right)(x-2)^2(x-3)^2.$$

- Then, for each root $x = x_0$, we compute $\gcd(P(x_0, y), Q(x_0, y))$:
  1. $\mathrm{gcd(P, Q, x = 2) = (y - 2)(y - 3)}$.
  2. $\mathrm{gcd(P, Q, x = 3) = (y - 2)(y - 3)}$.
  3. $\mathrm{gcd(P, Q, x^2 - x + 4 = 0) = (2\,x - 1)\,y - 7 - x}$.
- One major trouble occurs: for roots defined symbolically such as those of $x^2 - x + 4$, we are repeating (modulo $x^2 - x + 4 = 0$) the computations performed to get $\mathrm{res}(P, Q, y)$.
- Solution: we store all the computations performed to get $\mathrm{res}(P, Q, y)$ and use the algorithm of (Xin Li, Moreno Maza & Wei Pan, ISSAC 2009) to deduce (at essentially no costs) the GCDs.

# Solving bivariate systems through subresultants

- Let $P, Q \in \mathbf{k}[x < y]$ with $\deg(P, y) \geq \deg(Q, y) =: q > 0$.
- The subresultant chain of $(P, Q)$ is a sequence $S_0, S_1, S_2, \ldots, S_{q-1}$ of polynomials in $\mathbf{k}[x < y]$.
- For each $d = 0 \cdots q - 1$ the polynomial $S_d$ is either null or has degree in $y$ at most $d$
- The polynomial $S_0$ is $\mathrm{res}(P, Q, y)$ and has degree in $x$ at most $\delta = \deg(P, y)\deg(Q, x) + \deg(Q, y)\deg(P, x)$
- The polynomials $S_0, S_1, S_2, \ldots, S_{q-1}$ can be computed as follows:
  1. Evaluate $P, Q$ at $x = x_0, \ldots, x_\delta$ pairwise different values not cancelling both $\mathrm{lc}(P, y)$ and $\mathrm{lc}(Q, y)$.
  2. For each $x_i$ compute the subresultants of $P(x_i, y)$ and $Q(x_i, y)$ leading to evaluations of $S_0, S_1, s_2, \ldots, S_{q-1}$ at $x = x_i$.
  3. Interpolate the necessary $S_0, S_1, S_2, \ldots, S_{q-1}$.
- Therefore, computations are essentially univariate allowing the use of FFT-based polynomial arithmetic.

# Solving bivariate systems through subresultants

- Let $P, Q \in \mathbf{k}[x < y]$ with $\deg(P, y) \geq \deg(Q, y) =: q > 0$.
- The subresultant chain of $(P, Q)$ is a sequence $S_0, S_1, S_2, \ldots, S_{q-1}$ of polynomials in $\mathbf{k}[x < y]$.
- For each $d = 0 \cdots q - 1$ the polynomial $S_d$ is either null or has degree in $y$ at most $d$
- The polynomial $S_0$ is $\mathrm{res}(P, Q, y)$ and has degree in $x$ at most $\delta = \deg(P, y)\deg(Q, x) + \deg(Q, y)\deg(P, x)$
- The polynomials $S_0, S_1, S_2, \ldots, S_{q-1}$ can be computed as follows:
  1. Evaluate $P, Q$ at $x = x_0, \ldots, x_\delta$ pairwise different values not cancelling both $\mathrm{lc}(P, y)$ and $\mathrm{lc}(Q, y)$.
  2. For each $x_i$ compute the subresultants of $P(x_i, y)$ and $Q(x_i, y)$ leading to evaluations of $S_0, S_1, s_2, \ldots, S_{q-1}$ at $x = x_i$.
  3. Interpolate the necessary $S_0, S_1, S_2, \ldots, S_{q-1}$.
- Therefore, computations are essentially univariate allowing the use of FFT-based polynomial arithmetic.

# Solving bivariate systems through subresultants

- Let $P, Q \in \mathbf{k}[x < y]$ with $\deg(P, y) \geq \deg(Q, y) =: q > 0$.
- The subresultant chain of $(P, Q)$ is a sequence $S_0, S_1, S_2, \ldots, S_{q-1}$ of polynomials in $\mathbf{k}[x < y]$.
- For each $d = 0 \cdots q - 1$ the polynomial $S_d$ is either null or has degree in $y$ at most $d$
- The polynomial $S_0$ is $\mathrm{res}(P, Q, y)$ and has degree in $x$ at most $\delta = \deg(P, y)\deg(Q, x) + \deg(Q, y)\deg(P, x)$
- The polynomials $S_0, S_1, S_2, \ldots, S_{q-1}$ can be computed as follows:
  1. Evaluate $P, Q$ at $x = x_0, \ldots, x_\delta$ pairwise different values not cancelling both $\mathrm{lc}(P, y)$ and $\mathrm{lc}(Q, y)$.
  2. For each $x_i$ compute the subresultants of $P(x_i, y)$ and $Q(x_i, y)$ leading to evaluations of $S_0, S_1, s_2, \ldots, S_{q-1}$ at $x = x_i$.
  3. Interpolate the necessary $S_0, S_1, S_2, \ldots, S_{q-1}$.
- Therefore, computations are essentially univariate allowing the use of FFT-based polynomial arithmetic.

# Solving bivariate systems through subresultants

- Let $P, Q \in \mathbf{k}[x < y]$ with $\deg(P, y) \geq \deg(Q, y) =: q > 0$.
- The subresultant chain of $(P, Q)$ is a sequence $S_0, S_1, S_2, \ldots, S_{q-1}$ of polynomials in $\mathbf{k}[x < y]$.
- For each $d = 0 \cdots q - 1$ the polynomial $S_d$ is either null or has degree in $y$ at most $d$
- The polynomial $S_0$ is $\mathrm{res}(P, Q, y)$ and has degree in $x$ at most $\delta = \deg(P, y)\deg(Q, x) + \deg(Q, y)\deg(P, x)$
- The polynomials $S_0, S_1, S_2, \ldots, S_{q-1}$ can be computed as follows:
  1. Evaluate $P, Q$ at $x = x_0, \ldots, x_\delta$ pairwise different values not cancelling both $\mathrm{lc}(P, y)$ and $\mathrm{lc}(Q, y)$.
  2. For each $x_i$ compute the subresultants of $P(x_i, y)$ and $Q(x_i, y)$ leading to evaluations of $S_0, S_1, s_2, \ldots, S_{q-1}$ at $x = x_i$.
  3. Interpolate the necessary $S_0, S_1, S_2, \ldots, S_{q-1}$.
- Therefore, computations are essentially univariate allowing the use of FFT-based polynomial arithmetic.

# Solving bivariate systems through subresultants

- Let $P, Q \in \mathbf{k}[x < y]$ with $\deg(P, y) \geq \deg(Q, y) =: q > 0$.
- The subresultant chain of $(P, Q)$ is a sequence $S_0, S_1, S_2, \ldots, S_{q-1}$ of polynomials in $\mathbf{k}[x < y]$.
- For each $d = 0 \cdots q - 1$ the polynomial $S_d$ is either null or has degree in $y$ at most $d$
- The polynomial $S_0$ is $\mathrm{res}(P, Q, y)$ and has degree in $x$ at most $\delta = \deg(P, y)\deg(Q, x) + \deg(Q, y)\deg(P, x)$
- The polynomials $S_0, S_1, S_2, \ldots, S_{q-1}$ can be computed as follows:
  1. Evaluate $P, Q$ at $x = x_0, \ldots, x_\delta$ pairwise different values not cancelling both $\mathrm{lc}(P, y)$ and $\mathrm{lc}(Q, y)$.
  2. For each $x_i$ compute the subresultants of $P(x_i, y)$ and $Q(x_i, y)$ leading to evaluations of $S_0, S_1, s_2, \ldots, S_{q-1}$ at $x = x_i$.
  3. Interpolate the necessary $S_0, S_1, S_2, \ldots, S_{q-1}$.
- Therefore, computations are essentially univariate allowing the use of FFT-based polynomial arithmetic.

# Solving bivariate systems through subresultants

- Let $P, Q \in \mathbf{k}[x < y]$ with $\deg(P, y) \geq \deg(Q, y) =: q > 0$.
- The subresultant chain of $(P, Q)$ is a sequence $S_0, S_1, S_2, \ldots, S_{q-1}$ of polynomials in $\mathbf{k}[x < y]$.
- For each $d = 0 \cdots q - 1$ the polynomial $S_d$ is either null or has degree in $y$ at most $d$
- The polynomial $S_0$ is $\mathrm{res}(P, Q, y)$ and has degree in $x$ at most $\delta = \deg(P, y)\deg(Q, x) + \deg(Q, y)\deg(P, x)$
- The polynomials $S_0, S_1, S_2, \ldots, S_{q-1}$ can be computed as follows:
  1. Evaluate $P, Q$ at $x = x_0, \ldots, x_\delta$ pairwise different values not cancelling both $\mathrm{lc}(P, y)$ and $\mathrm{lc}(Q, y)$.
  2. For each $x_i$ compute the subresultants of $P(x_i, y)$ and $Q(x_i, y)$ leading to evaluations of $S_0, S_1, s_2, \ldots, S_{q-1}$ at $x = x_i$.
  3. Interpolate the necessary $S_0, S_1, S_2, \ldots, S_{q-1}$.
- Therefore, computations are essentially univariate allowing the use of FFT-based polynomial arithmetic.

# Plan

# Background

## Motivation

- FFTs over finite fields is at the core of asymptotically fast polynomial arithmetic.
- Most FFTs on GPUs are for complex numbers, such as NVIDIA CUFFT library.

## 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, $30 \times 8$ cores, memory interface width 512 bits

# Discrete Fourier Transform (DFT)

## Definition

Given a primitive $n$-th root of unity $\omega$ (i.e. $\omega^{n/2} = -1$), and

$$f(t) = x_0 + x_1 t + \cdots + x_{n-1} t^{n-1},$$

$\mathrm{DFT}_n^{\omega}(f)$ is $\mathbf{y} = (y_0, \ldots, y_{n-1})$ with $y_k = f(\omega^k)$ for $0 \leq k < n$. As a matrix-vector product, it is

$$\mathbf{y} = \mathrm{DFT}_n \mathbf{x}, \quad \mathrm{DFT}_n = [\omega^{k\ell}]_{0 \leq k, \ell < n}. \tag{1}$$

## Example

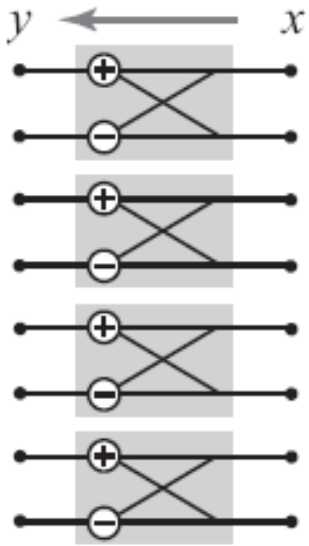
$$\begin{cases} y_0 &= x_0 + x_1 \\ y_1 &= x_0 - x_1 \end{cases} \iff \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

That is, $\mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$.

# Extracting parallelism from structural formulas

$I_n \otimes A$: block parallelism

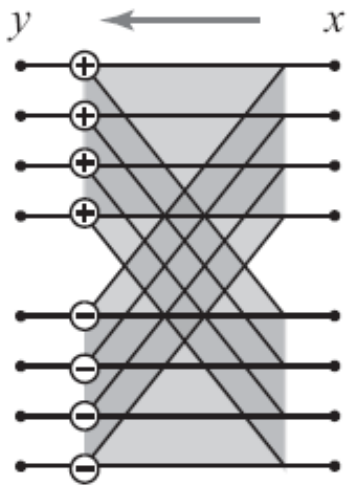
$$I_4 \otimes \mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 & & & & & & \\ 1 & -1 & & & & & & \\ & & 1 & 1 & & & & \\ & & 1 & -1 & & & & \\ & & & & 1 & 1 & & \\ & & & & 1 & -1 & & \\ & & & & & & 1 & 1 \\ & & & & & & 1 & -1 \end{bmatrix}$$



# Extracting 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.

# Plan

1. Solving Bivariate Polynomial Systems
2. FFT over Finite Fields on the GPU
3. Solving Bivariate Polynomial Systems over Finite Fields on the GPU

# Solving bivariate polynomial systems

## Recall of our bivariate system solving strategy

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 deduced from a so-called **Subresultant Chain Computation** which can be computed via evaluation/interpolation, thus **via FFT.**

# Subresultant chain constructions via FFT

Let $P, Q \in \mathbf{k}[x, y]$ such that $p = \deg(P, y) \geq q = \deg(Q, y) > 1$, where $\mathbf{k}$ is a **finite field**, for instance $\mathbb{Z}/9001\mathbb{Z}$.

1. Compute degree bound $n = 2^\ell$ with
$$\deg_x \operatorname{res}(P, Q, y) < n.$$
2. Evaluate $P$, $Q$ at $x = \omega^j$ for $j = 0 \cdots n-1$ by calling
$$I_{p+1} \otimes \mathrm{DFT}_n^\omega(P) \text{ and } I_{q+1} \otimes \mathrm{DFT}_n^\omega(Q),$$
with a **well-chosen** $n$-primitive root of unity $\omega$.
3. Compute univariate subresultant chains
$$R_j = \mathrm{SubResChain}(P_j, Q_j, y)$$
for all evaluation images $(P_j, Q_j)$, which essentially boils down to a sequence of **univariate (pseudo)-divisions**.
4. These images form a data-structure that we call the **SCube**:
   - it is built and stored on the GPU,
   - the host runs code which queries the SCube.

# Subresultant chain constructions via FFT

Let $P, Q \in \mathbf{k}[x, y]$ such that $p = \deg(P, y) \geq q = \deg(Q, y) > 1$, where $\mathbf{k}$ is a **finite field**, for instance $\mathbb{Z}/9001\mathbb{Z}$.

1. Compute degree bound $n = 2^\ell$ with

$$\deg_x \operatorname{res}(P, Q, y) < n.$$

2. Evaluate $P$, $Q$ at $x = \omega^j$ for $j = 0 \cdots n-1$ by calling

$$I_{p+1} \otimes \mathrm{DFT}_n^\omega(P) \text{ and } I_{q+1} \otimes \mathrm{DFT}_n^\omega(Q),$$

   with a **well-chosen** $n$-primitive root of unity $\omega$.

3. Compute univariate subresultant chains

$$R_j = \mathrm{SubResChain}(P_j, Q_j, y)$$

   for all evaluation images $(P_j, Q_j)$, which essentially boils down to a sequence of **univariate (pseudo)-divisions**.

4. These images form a data-structure that we call the **SCube**:
   - it is built and stored on the GPU,
   - the host runs code which queries the SCube.

# Subresultant chain constructions via FFT

Let $P, Q \in \mathbf{k}[x, y]$ such that $p = \deg(P, y) \geq q = \deg(Q, y) > 1$, where $\mathbf{k}$ is a **finite field**, for instance $\mathbb{Z}/9001\mathbb{Z}$.

1. Compute degree bound $n = 2^\ell$ with

$$\deg_x \operatorname{res}(P, Q, y) < n.$$

2. Evaluate $P$, $Q$ at $x = \omega^j$ for $j = 0 \cdots n-1$ by calling

$$I_{p+1} \otimes \mathrm{DFT}_n^\omega(P) \text{ and } I_{q+1} \otimes \mathrm{DFT}_n^\omega(Q),$$

   with a **well-chosen** $n$-primitive root of unity $\omega$.

3. Compute univariate subresultant chains

$$R_j = \mathrm{SubResChain}(P_j, Q_j, y)$$

   for all evaluation images $(P_j, Q_j)$, which essentially boils down to a sequence of **univariate (pseudo)-divisions**.

4. These images form a data-structure that we call the **SCube**:
   - it is built and stored on the GPU,
   - the host runs code which queries the SCube.

# Subresultant chain constructions via FFT

Let $P, Q \in \mathbf{k}[x, y]$ such that $p = \deg(P, y) \geq q = \deg(Q, y) > 1$, where $\mathbf{k}$ is a **finite field**, for instance $\mathbb{Z}/9001\mathbb{Z}$.

1. Compute degree bound $n = 2^\ell$ with
$$\deg_x \operatorname{res}(P, Q, y) < n.$$
2. Evaluate $P$, $Q$ at $x = \omega^j$ for $j = 0 \cdots n-1$ by calling
$$I_{p+1} \otimes \mathrm{DFT}_n^\omega(P) \text{ and } I_{q+1} \otimes \mathrm{DFT}_n^\omega(Q),$$
with a **well-chosen** $n$-primitive root of unity $\omega$.
3. Compute univariate subresultant chains
$$R_j = \mathrm{SubResChain}(P_j, Q_j, y)$$
for all evaluation images $(P_j, Q_j)$, which essentially boils down to a sequence of **univariate (pseudo)-divisions**.
4. These images form a data-structure that we call the **SCube**:
   - it is built and stored on the GPU,
   - the host runs code which queries the SCube.

# Subresultant chain via FFT: implementation issues

## Different strategies

- Issues with the direct FFT method:
  - Fourier prime limitation,
  - a valid grid is required,
  - translations are tried a few times.
- Subproduct tree techniques (indirect use of FFT, less efficient) are back-up solutions.

## Construction of the SCube on the GPU

- Coarse-grained construction (always works)
- Fine-grained construction (requires some genericity assumption)

# Subresultant chain via FFT: implementation issues

## Different strategies

- Issues with the direct FFT method:
  - Fourier prime limitation,
  - a valid grid is required,
  - translations are tried a few times.
- Subproduct tree techniques (indirect use of FFT, less efficient) are back-up solutions.

## Construction of the SCube on the GPU

- Coarse-grained construction (always works)
- Fine-grained construction (requires some genericity assumption)

# Approach (I) : coarse-grained implementation

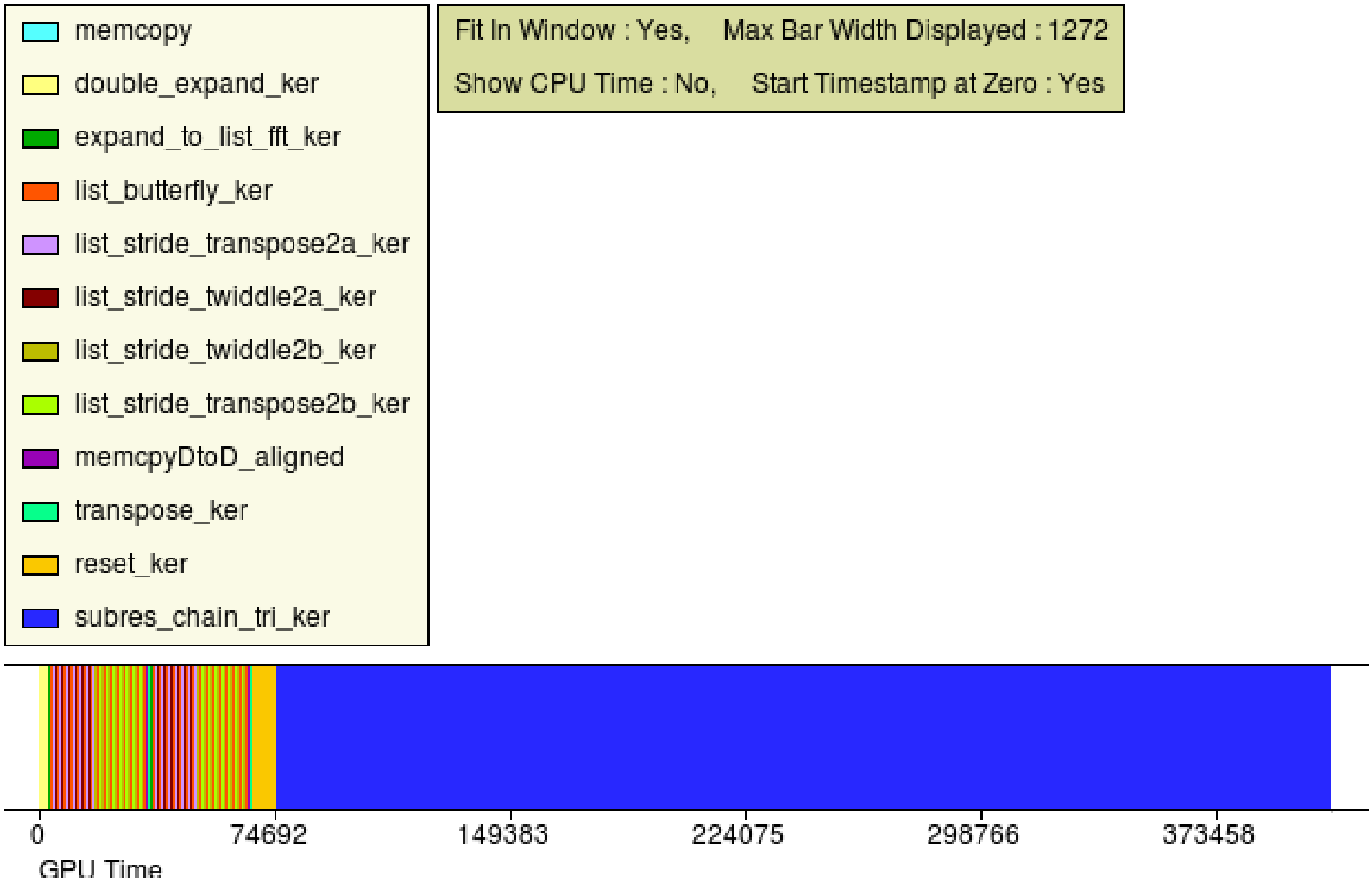
## Main idea

Each CUDA thread runs a univariate subresultant algorithm.

## Remarks

- Simple and always works.
- Still enough threads when $n$ is big.
- Unfavoured memory access pattern to the GPU memory: threads in a thread wrap access different memory regions in an irregular fashion.

# Profiling coarse-grained implementation



# Approach (II) : fine-grained implementation

## Main idea

Break a list of univariate subresultant computations into a sequence of **lists of** univariate polynomial (pseudo)-divisions.

## Remarks

- Tricky to implement.
- Require the following further assumption:
  **The degree sequences of all images $(P_j, Q_j)$ are the same.**

# Break pseudo-divisions

## Example

Let $f = a_3x^3 + a_2x^2 + a_1x + a_0$ and $g = b_2x^2 + b_1x + b_0$. To obtain the pseudo-remainder $\mathrm{prem}(f, -g, x)$ of $f$ and $-g$, we compute

1. $h_2 = -b_2 f + a_3 x g = c_2x^2 + c_1x + c_0$,
2. $h_1 = -b_2 h_2 + c_2 g = d_1x + b_0$.

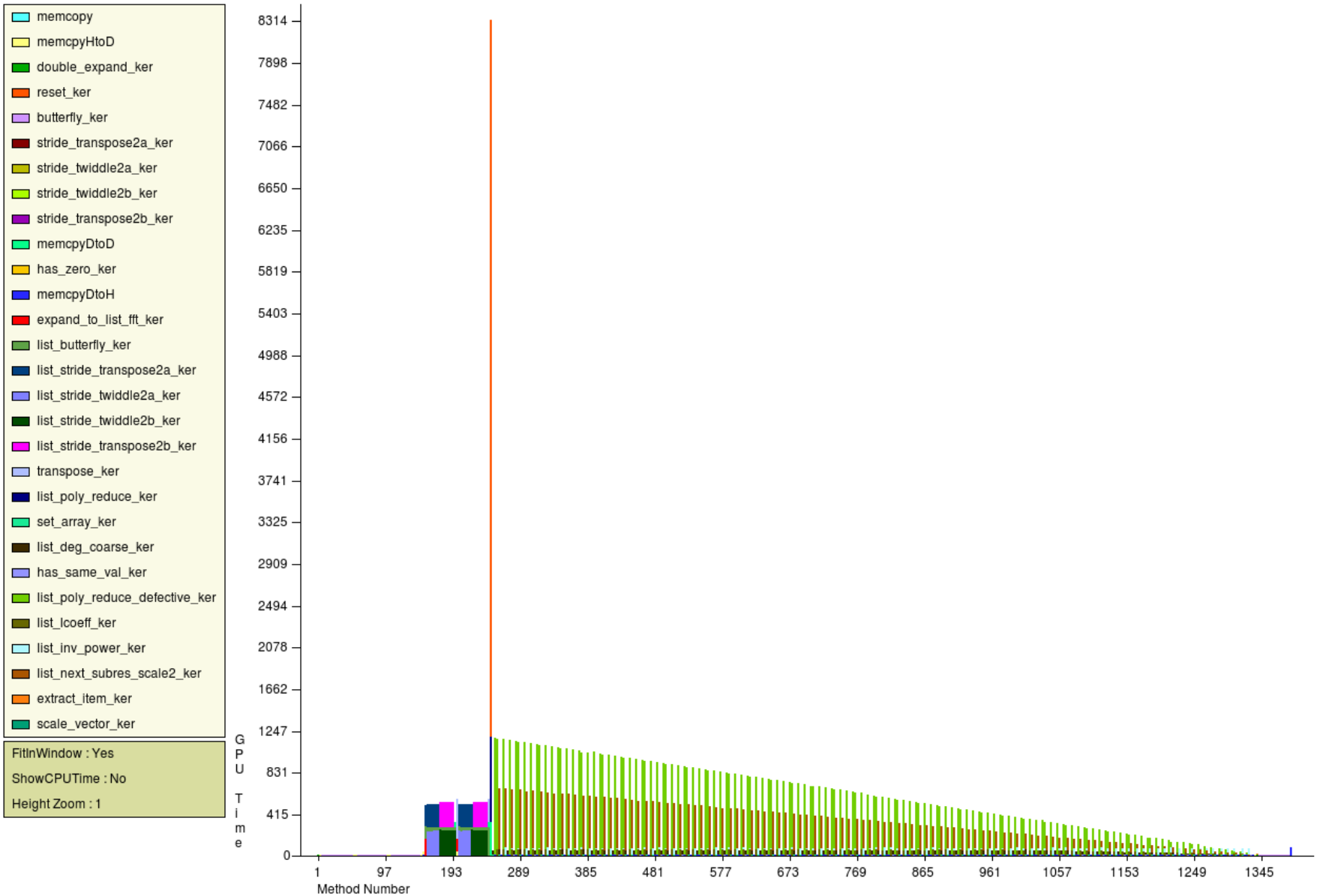
Alternatively, we compute

$$\text{(S1)} \quad c_2 = \begin{vmatrix} a_3 & a_2 \\ b_2 & b_1 \end{vmatrix} \quad c_1 = \begin{vmatrix} a_3 & a_1 \\ b_2 & 0 \end{vmatrix} \quad c_0 = \begin{vmatrix} a_3 & a_0 \\ b_2 & 0 \end{vmatrix}$$

$$\text{(S2)} \quad d_1 = \begin{vmatrix} c_2 & c_1 \\ b_2 & b_1 \end{vmatrix} \quad d_0 = \begin{vmatrix} c_2 & c_0 \\ b_2 & b_0 \end{vmatrix}$$

One can do a list of pseudo-divisions with the same input degrees.

# 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 |

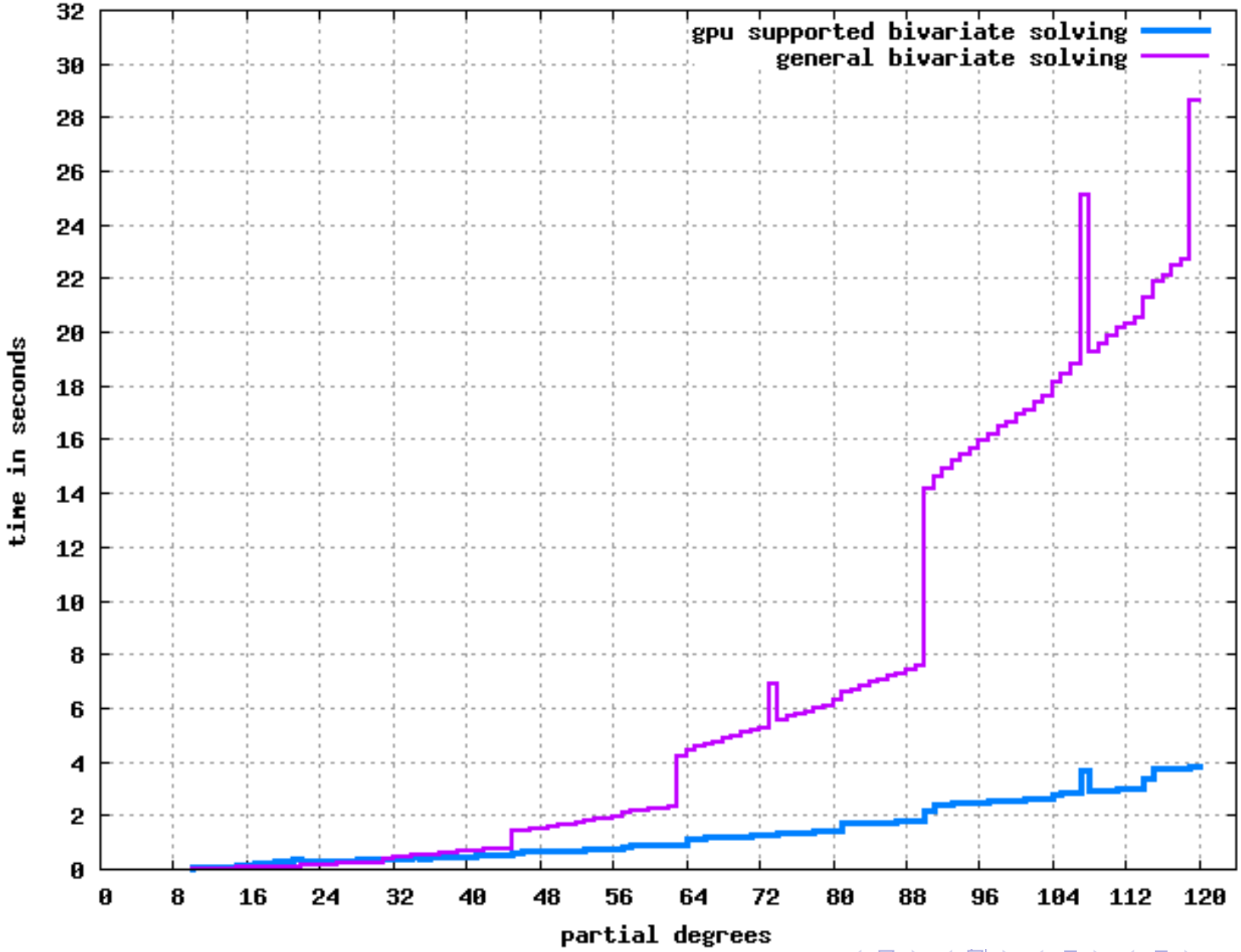
Bivariate dense polynomials of total degree $d$.

| $d$ | $t_0$ | $t_1$ | $t_1/t_0$ |
|---|---|---|---|
| 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 |

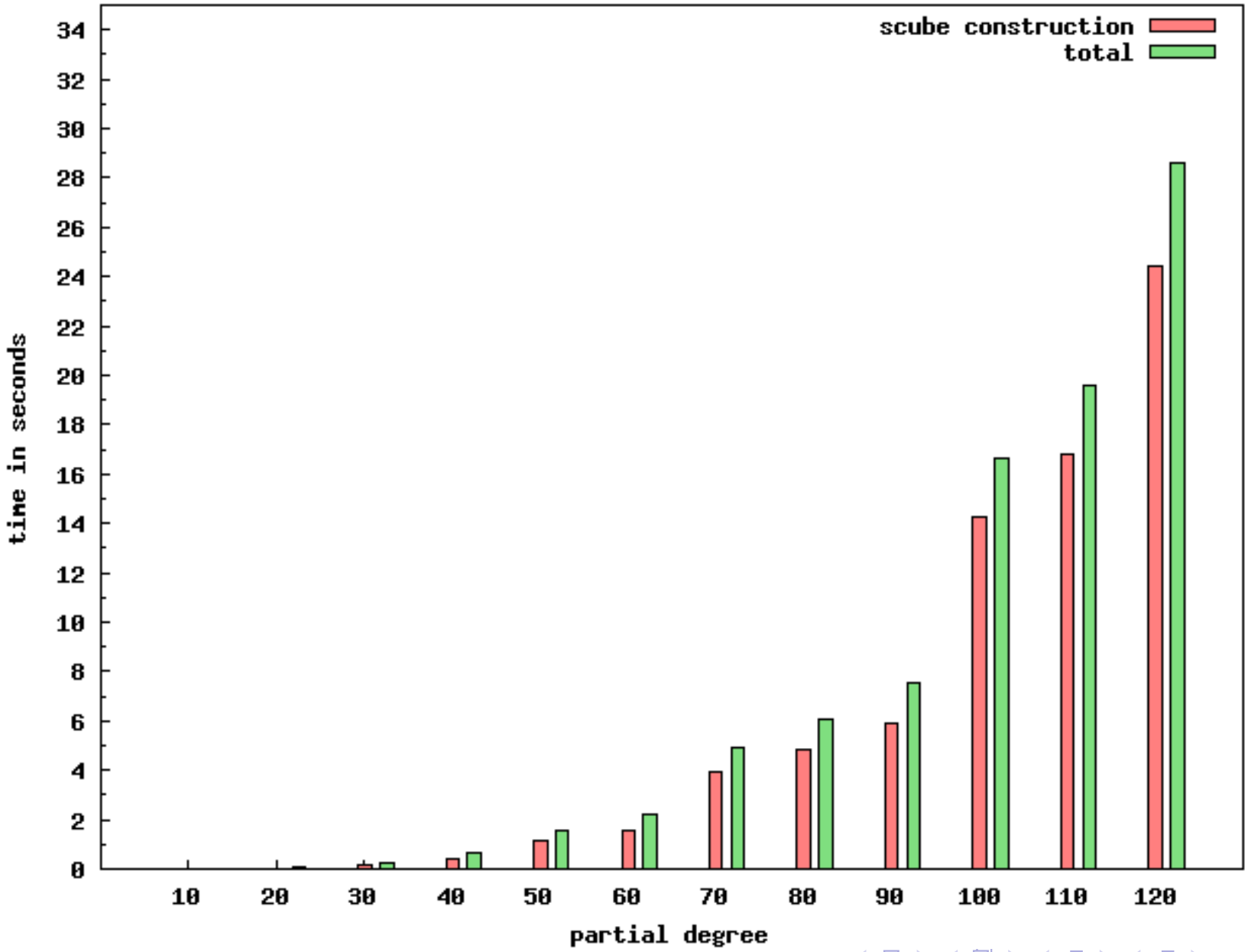
Trivariate dense polynomials of total degree $d$.

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

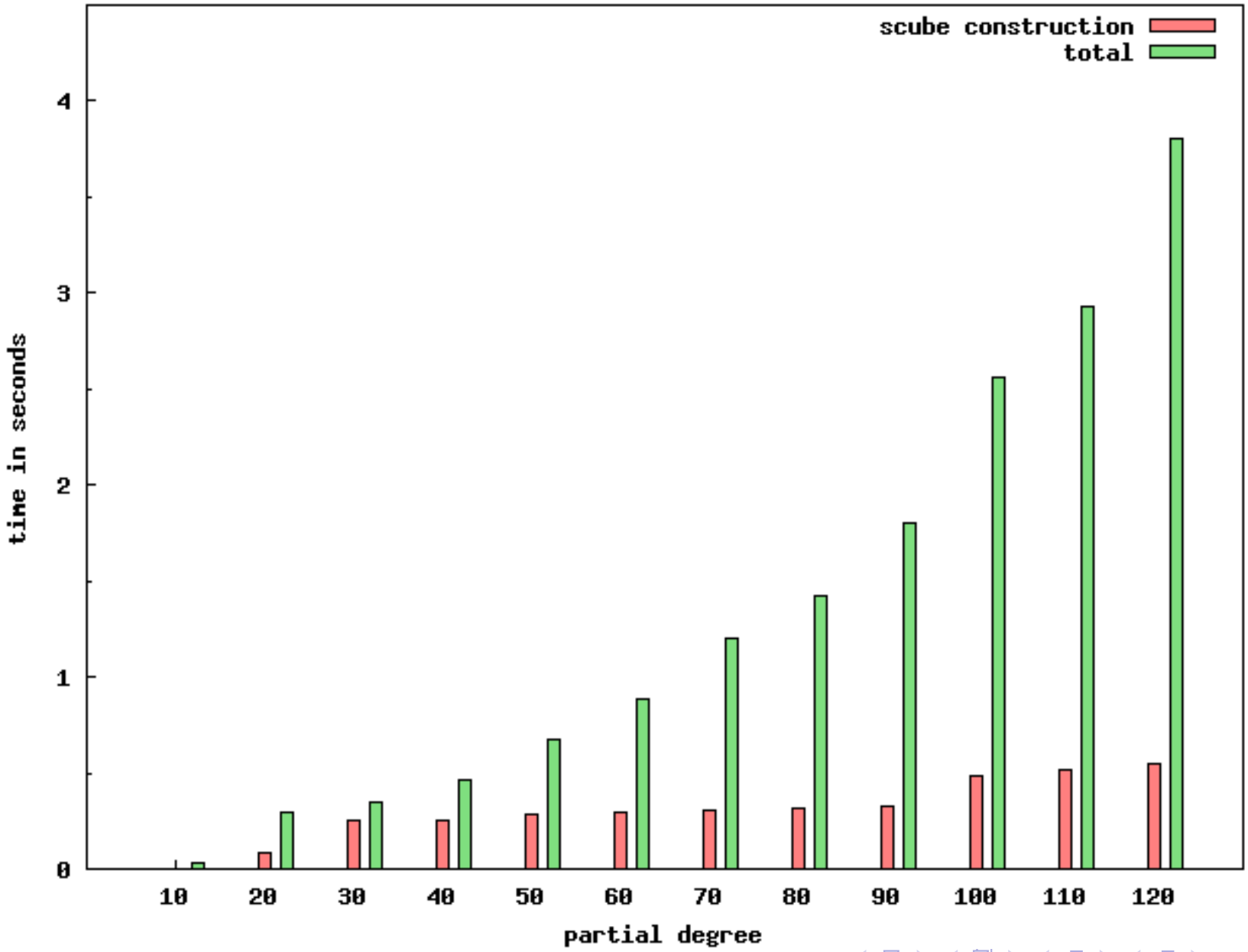
# Bivariate solver



# Bivariate solver on the CPU



# Bivariate solver on the GPU



# Solving bivariate systems in seconds

| $d$ | $t_0$(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 CPU 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.

# Subresultant chain computation