Evaluating polynomials in several variables and their derivatives on a GPU computing processor

Genady Yoffe
joint work with Jan Verschelde

University of Illinois at Chicago
Department of Mathematics, Statistics, and Computer Science
http://www.math.uic.edu/~gyoffe
gyoffe2@uic.edu

LANS Seminar, MCS, Argonne National Laboratory
May 23rd, 2012
A polynomial in $n$ variables $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ consists of a vector of nonzero complex coefficients with corresponding exponents in $A$:

$$f(\mathbf{x}) = \sum_{\mathbf{a} \in A} c_{\mathbf{a}} \mathbf{x}^{\mathbf{a}}, \quad c_{\mathbf{a}} \in \mathbb{C} \setminus \{0\}, \quad \mathbf{x}^{\mathbf{a}} = x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n}.$$ 

Given is a system $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ and some point $\mathbf{z} \in \mathbb{C}^n$.

The problem is to evaluate $\mathbf{f}$ and its Jacobian matrix $J_{\mathbf{f}}$ at $\mathbf{z}$, i.e.: to compute the vector $\mathbf{f}(\mathbf{z})$ and the matrix $J_{\mathbf{f}}(\mathbf{z})$.

We refer to $n$ as to the system’s dimension.
We consider evaluation and differentiation of polynomial systems in context of Newton’s method used as the corrector in path tracking.

Thus we need to evaluate the system an its Jacobian several thousand times.

**Advantage:** system’s information is transferred to the global memory of the card only once.
For large polynomial systems in many (20-100) variables and high degrees (yet we consider total degrees of monomials to be $\leq \frac{n}{2}$):

- the cost of polynomial evaluation and differentiation often dominates the linear algebra of Newton’s method; and
- the double precision as available in standard hardware is often insufficient to guarantee accurate results.

**Goal:** offset the extra cost of extended precision by parallel computing.

- We implemented before a multithreaded path tracker for a multicore workstation.
- The presented implementation is designed towards obtaining a GPU version of a path tracker.
settings: sparse polynomials; regularity assumptions for benchmarks

- We consider *sparse* polynomials. Relatively few monomials. Typically we take $\Theta(n)$ monomials in a polynomial.
- For dense polynomials use Horner scheme.
- Sparse polynomials appear in most applications.

We consider first systems with

- fixed number of variables $k$ in the monomials;
- fixed maximal degree $d$ up to which each of the variables appears in the system
- fixed number $m$ of monomials in each of the polynomials

**Irregularity in variables positions and exponents.**
quad double precision

A quad double is an unevaluated sum of 4 doubles, improves working precision from $2.2 \times 10^{-16}$ to $2.4 \times 10^{-63}$.


Predictable overhead: working with *double double* is of the same cost as working with complex numbers. Simple memory management.

The QD library has been ported to the GPU by

computers and compilers

Hardware:
- HP Z800 workstation running Red Hat Enterprise Linux 6.1
  The CPU is an Intel Xeon X5690 at 3.47 Ghz.
- The processor clock of the NVIDIA Tesla C2050 Computing Processor runs at 1147 Mhz. The graphics card has 14 multiprocessors, each with 32 cores, for a total of 448 cores.

As the clock speed of the GPU is a third of the clock speed of the CPU, we hope to achieve a double digit speedup.

Compilers:
- Code written in C++ using `gcc` version 4.4.6.
- NVIDIA CUDA compiler driver `nvcc`, release 4.0, V0.2.1221.
Polynomials are linear combinations of monomials \( x^a = x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n} \).

Separating monomial products from products of variables:

\[
x^a = \left( x_{i_1}^{a_{i_1}-1} x_{i_2}^{a_{i_2}-1} \cdots x_{i_k}^{a_{i_k}-1} \right) \star \left( x_{i_1} x_{i_2} \cdots x_{i_k} \right),
\]

for \( a_{im} \geq 1, m = 1, 2, \ldots, k, 1 \leq i_1 < i_2 < \cdots < i_k \leq n \).

Evaluating and differentiating \( x^a \) in three steps:

1. compute the monomial products \( x_{i_1}^{a_{i_1}-1} x_{i_2}^{a_{i_2}-1} \cdots x_{i_k}^{a_{i_k}-1} \)
2. compute \( x_{i_1} x_{i_2} \cdots x_{i_k} \) and its gradient
3. multiply the evaluated \( x_{i_1} x_{i_2} \cdots x_{i_k} \) and the components of its gradient with the evaluated monomial products
computing monomial products \( x_{i_1}^{a_{i_1} - 1} x_{i_2}^{a_{i_2} - 1} \cdots x_{i_k}^{a_{i_k} - 1} \)

first kernel

To evaluate \( x_1^3 x_2^7 x_3^2 \) and its derivatives, we first evaluate the factor \( x_1^2 x_2^6 x_3 \) and then multiply this factor with all derivatives of \( x_1 x_2 x_3 \).

Because \( x_1^2 x_2^6 x_3 \) is common to the evaluated monomial and all its derivatives, we call \( x_1^2 x_2^6 x_3 \) a \textit{common factor}.

The kernel to compute common factors operates in two stages:

1. Each of the first \( n \) threads of a thread block computes sequentially powers from the 2nd to the \((d - 1)\)th of one of the \( n \) variables.
2. Each of the threads of a block computes a common factor for one of the monomials of the system, as a product of \( k \) quantities computed at the first stage of the kernel.

The threads of a block are synchronized between the two stages of the kernel.
optimization

- The precomputed powers of variables are stored in shared memory: the \((i, j)\)th element stores \(x^i_j\), minimizing bank conflicts.

- The positions and exponents of variables in monomials are stored in two one-dimensional arrays in constant memory.
common factor calculation

constant memory

thread computes:

\[ x_1^5 \times x_3^4 \times \cdots \]

\[ x_1^5 \times x_3^4 \times \cdots \]

\[ x_1^2 \times x_3^2 \times x_n^2 \]

\[ x_1^3 \times x_3^3 \times x_n^3 \]

\[ x_1^4 \times x_3^4 \times x_n^4 \]

\[ \cdots \]

\[ \cdots \]

\[ x_1^{d-1} \times x_3^{d-1} \times x_n^{d-1} \]

POSITIONS

\[ \cdots 1 \ 3 \ \cdots \]

EXPOENTS

\[ \cdots 5 \ 4 \ \cdots \]

POWERs

\[ x_1^2 \ x_2^2 \ x_3^2 \ \cdots \ x_n^2 \]

\[ x_1^3 \ x_2^3 \ x_3^3 \ \cdots \ x_n^3 \]

\[ x_1^4 \ x_2^4 \ x_3^4 \ x_n^4 \]

\[ \cdots \]

\[ \cdots \]

\[ x_1^{d-1} \ x_2^{d-1} \ x_3^{d-1} \ \cdots \ x_n^{d-1} \]
working dimensions

1 Need at least 1,000 monomials (1 monomial per one thread) to occupy well all the 14 multiprocessors of the card, so to hide long latency operations.

2 Try to keep the number of terms in a polynomial approximately equal to dimension.

Thus \( n \geq 30 \), so the system would have at least \( 30 \times 30 \approx 1000 \) monomials.

On the other hand

1 The capacity of the constant memory is 65,536 bytes.

2 When the number of variables in a monomial \( k \leq n/2 \) need \( n/2 \) bytes to store positions of variables in a monomial, and \( n/2 \) bytes to store exponents of variables in a monomial.

Thus \( n \leq 40 \), since for \( n = 40 \) the size of the positions and degrees information would take \( n/2 \times 1600 + n/2 \times 1600 = 64,000 \) bytes of shared memory, which is almost its full capacity.

Conclusion. Thus working dimensions for our implementation are between 30 and 40.
more efficient compression of monomial information?

**Goal:** want to introduce more compact encodings for storing positions and exponents of variables in constant memory.

**Rationale:** This would allow to work with higher dimensions.

- More efficient compression might introduce some branching within a warp, as threads will be decoding the information to know the actual positions and exponents of variables.

- Computations which are to follow the decodings, where threads of a warp again will follow one execution path, suppose to dominate decodings in time.

Indeed decodings would employ an integer arithmetic, while computations to follow eventually will be expensive multiprecision multiplications.

**Conclusion.** Thus with new ways of compression of monomial information, and employed multiprecision, we hope to increase working dimensions of our implementation.
reverse mode AD for $x_1 \times x_2 \times x_3 \times x_4 \times x_5$, $k = 5$

1. We first compute $\psi$-s by $(k - 2)$ multiplications:
   
   $\psi_1 = (x_1)$,
   
   $\psi_2 = \psi_1 \times (x_2) = (x_1) \times (x_2)$,
   
   $\psi_3 = \psi_2 \times (x_3) = (x_1 \times x_2) \times (x_3)$,
   
   $\psi_4 = \psi_3 \times (x_4) = (x_1 \times x_2 \times x_3) \times (x_4)$;

2. and $\varphi$-s by $(k - 2)$ multiplications:

   $\varphi_1 = (x_5)$,
   
   $\varphi_2 = \varphi_1 \times (x_4) = (x_5) \times (x_4)$,
   
   $\varphi_3 = \varphi_2 \times x_3 = (x_5 \times x_4) \times (x_3)$,
   
   $\varphi_4 = \varphi_3 \times (x_2) = (x_5 \times x_4 \times x_3) \times (x_2)$.

3. Eventually by $(k - 2)$ multiplications we compute $\omega$-s:

   $\omega_1 = \psi_4 = (x_1 \times x_2 \times x_3 \times x_4)$,
   
   $\omega_2 = \psi_3 \times \varphi_1 = (x_1 \times x_2 \times x_3) \times (x_5)$,
   
   $\omega_3 = \psi_2 \times \varphi_2 = (x_1 \times x_2) \times (x_5 \times x_4)$,
   
   $\omega_4 = \psi_1 \times \varphi_3 = (x_1) \times (x_5 \times x_4 \times x_3)$,
   
   $\omega_5 = \varphi_4 = (x_5 \times x_4 \times x_3 \times x_2)$.

The total # of multiplications is $(k - 2) + (k - 2) + (k - 2) = 3k - 6$. 

Genady Yoffe (UIC) | Polynomial Evaluation on a GPU
Example: A thread of the second kernel computes the term 
\[ \beta = c x_1^3 x_2^7 x_3^4 x_4^5 \]
and its derivatives by 
\[ 5k - 4 = 5 \times 4 - 4 = 16 \]
multiplications, using 
\[ k + 1 = 5 \]
shared memory locations and one local variable.

Here \( \gamma := \frac{1}{c} \beta = x_1^3 x_2^7 x_3^4 x_4^5 \)
is the corresponding monomial, 
\( \alpha := x_1^2 x_2^6 x_3^3 x_4^4 \) (the common factor), and 
\( s := x_1 x_2 x_3 x_4 \) (the speelpenning product).

Note that the coefficients (3c), (7c), (4c), (5c) are precomputed, only explicit \(*\)-s stay for performed multiplications.
Work of one thread is demonstrated.

To compute the derivatives of $s = x_1 x_2 x_3 x_4$,

- $Q$ stores the backward product, and
- the $i$th partial derivative of $S$ is stored in memory location $L_i$.

<table>
<thead>
<tr>
<th>$L_1$</th>
<th>$L_2$</th>
<th>$L_3$</th>
<th>$L_4$</th>
<th>$Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$x_1$</td>
<td>$x_1 * x_2$</td>
<td>$(x_1 x_2) * x_3$</td>
<td>$x_4$</td>
</tr>
<tr>
<td>$x_1$</td>
<td>$x_1$</td>
<td>$x_1 x_2$</td>
<td>$x_1 x_2 x_3$</td>
<td>$x_4 * x_3$</td>
</tr>
<tr>
<td>$x_1$</td>
<td>$x_1 * (x_3 x_4)$</td>
<td>$x_1 x_2 x_4$</td>
<td>$(x_4 x_3) * x_2$</td>
<td></td>
</tr>
<tr>
<td>$x_2 x_3 x_4$ ($= \frac{\partial s}{\partial x_1}$)</td>
<td>$x_1 x_3 x_4$ ($= \frac{\partial s}{\partial x_2}$)</td>
<td>$x_1 x_2 x_4$ ($= \frac{\partial s}{\partial x_3}$)</td>
<td>$x_1 x_2 x_3$ ($= \frac{\partial s}{\partial x_4}$)</td>
<td></td>
</tr>
</tbody>
</table>

The performed multiplications are marked by stars ($\ast$).
Obtaining the value of a monomial and its derivatives
second kernel continued

Given $s = x_1 x_2 x_3 x_4$ and its gradient, with $\alpha = x_1^2 x_2^6 x_3^3 x_4^4$ we evaluate $\beta = c x_1^3 x_2^7 x_3^4 x_4^5$ and its derivatives, denoting $\gamma = \frac{1}{c} \beta = x_1^3 x_2^7 x_3^4 x_4^5$.

$$
\begin{array}{c|c|c|c|c}
L_1 & L_2 & L_3 & L_4 & L_5 \\
\frac{\partial s}{\partial x_1} \star \alpha \left( = \frac{1}{3} \frac{\partial \gamma}{\partial x_1} \right) & \frac{\partial s}{\partial x_2} \star \alpha \left( = \frac{1}{7} \frac{\partial \gamma}{\partial x_2} \right) & \frac{\partial s}{\partial x_3} \star \alpha \left( = \frac{1}{4} \frac{\partial \gamma}{\partial x_3} \right) & \frac{\partial s}{\partial x_4} \star \alpha \left( = \frac{1}{5} \frac{\partial \gamma}{\partial x_4} \right) & \frac{1}{5} \frac{\partial \gamma}{\partial x_4} \star x_4 \left( = \gamma \right) \\
\frac{1}{3} \frac{\partial \gamma}{\partial x_1} & \frac{1}{7} \frac{\partial \gamma}{\partial x_2} & \frac{1}{4} \frac{\partial \gamma}{\partial x_3} & \frac{1}{5} \frac{\partial \gamma}{\partial x_4} & \gamma \star c \left( = \beta \right)
\end{array}
$$

Note that the coefficients $(3c), (7c), (4c), (5c)$ are precomputed. The performed multiplications marked by a stars($\star$).
shared memory capacity as limiting factor on working dimensions and precision

Can go up to \( \dim = 70 \) with double double arithmetic.

1. \( \left( \frac{n}{2} + 1 \right) \times 2 \times \text{sizeof(double double)} \leq \left( \frac{70}{2} + 1 \right) \times 2 \times 16 = 1152 \) bytes in shared memory. To treat 32 monomials by a block of 32 threads we would need then at most

\[
32 \times 1152 = 36864 \text{ bytes of shared memory.}
\]

2. For storing values of the variable we would need

\[
n \times \text{sizeof(complex double double)} \leq 70 \times 2 \times \text{sizeof(double double)} = 70 \times 2 \times 16 = 2240.
\]

3. Allocation both spaces in shared memory yet leave \((49,152-(36,864+2,240)) > 10,000 \) available bytes of shared memory.
We generate a system with random complex coefficients:
- a system of 32 polynomials,
- each monomial has 9 variables with nonzero power of at most 2,
- a varying number of monomials per polynomial: 22, 32, and 48 lead to 704, 1024, and 1536 monomials in the system.

Wall clock times and speedups for 100,000 evaluations:

<table>
<thead>
<tr>
<th>#monomials</th>
<th>Tesla C2050</th>
<th>1 CPU core</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>704</td>
<td>14.514 sec</td>
<td>1min 50.9 sec</td>
<td>7.60</td>
</tr>
<tr>
<td>1024</td>
<td>15.265 sec</td>
<td>2min 39.3 sec</td>
<td>10.44</td>
</tr>
<tr>
<td>1536</td>
<td>17.000 sec</td>
<td>3min 58.7 sec</td>
<td>14.04</td>
</tr>
</tbody>
</table>

At least 1000 monomials are needed for a modest speedup.
monomials of higher degrees

We generate a system with random complex coefficients:

- a system of 32 polynomials,
- each monomial has 16 variables with nonzero power $\leq 10$,
- a varying number of monomials per polynomial: 22, 32, and 48 lead to 704, 1024, and 1536 monomials in the system.

Wall clock times and speedups for 100,000 evaluations:

<table>
<thead>
<tr>
<th>#monomials</th>
<th>Tesla C2050</th>
<th>1 CPU core</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>704</td>
<td>19.068 sec</td>
<td>3min 16.9 sec</td>
<td>10.33</td>
</tr>
<tr>
<td>1024</td>
<td>20.800 sec</td>
<td>4min 43.3 sec</td>
<td>13.62</td>
</tr>
<tr>
<td>1536</td>
<td>21.763 sec</td>
<td>7min 05.8 sec</td>
<td>19.56</td>
</tr>
</tbody>
</table>

With higher degrees, we obtain higher speedups.
conclusions

We obtained modest speedups with our first code for the evaluation and differentiation of a polynomial system and its Jacobian matrix. On randomly generated systems, preliminary experiments show that

- for good occupancy at least 1000 monomials are needed,
- the size of constant memory limits more than 2000 monomials,
- speedups increase with higher degrees.

Ongoing and future work includes

- quality up factors with double double and quad double precision,
- adding a linear solver on the GPU implements Newton’s method,
- integration in the polynomial system solver of PHCpack,
- parallel reverse mode automatic differentiation on GPU.
our papers

Available at our web sites:


**New:** Accepted by IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-12) to be held May 21-25, 2012, Shanghai, China; will be published in the IPDPS-12 proceedings by IEEE CS