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

Jan Verschelde
joint work with Genady Yoffe

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

The 26th Parallel and Distributed Processing Symposium (IPDPS-12)
The 13th IEEE international Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-12)
21-25 May 2012, Shanghai, China
1 Problem Statement
- evaluating and differentiating polynomials in several variables
- quad double arithmetic on a graphics compute processor

2 Massively Parallel Polynomial Evaluation
- stages in the evaluation of a system and its Jacobian matrix
- computing monomial products from powers of variables
- evaluating and differentiating products of variables

3 Computational Experiments
- regularity assumptions on the input data
- computational results with the Tesla C2050
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 $A$:

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

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 $J_\mathbf{f}(\mathbf{z})$.

For large polynomial systems in many variables and high degrees:
- 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.
an arithmetic network for $x_1 \times x_2 \times x_3 \times x_4$


Evaluating $x_1 \times x_2 \times \cdots \times x_n$ and its gradient takes $3n - 5$ multiplications.
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 $\mathbf{x}^\mathbf{a} = x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n}$.

Separating monomial products from products of variables:

$$\mathbf{x}^\mathbf{a} = \left( x_{i_1}^{a_{i_1}-1} x_{i_2}^{a_{i_2}-1} \cdots x_{i_k}^{a_{i_k}-1} \right) \star \left( x_{j_1} x_{j_2} \cdots x_{j_\ell} \right),$$

for $a_{i_m} \geq 1$, $m = 1, 2, \ldots, k$, $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, and $1 \leq j_1 < j_2 < \cdots < j_\ell \leq n$, with $\ell \geq k$.

Evaluating and differentiating $\mathbf{x}^\mathbf{a}$ 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 $x_{j_1} x_{j_2} \cdots x_{j_\ell}$ and its gradient
3. multiply the evaluated $x_{j_1} x_{j_2} \cdots x_{j_\ell}$ and its gradient with the evaluated monomial products
computing monomial products $x_{i_1}^{a_{i_1}-1} x_{i_2}^{a_{i_2}-1} \cdots x_{i_k}^{a_{i_k}-1}$

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 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 precomputed powers of variables are stored in shared memory: the $(i,j)$th element stores $x_j^i$, minimizing bank conflicts.

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

thread computes:

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

constant memory

\[ \vdots \]

\[ \begin{array}{cccccc}
\text{Positions} \\
1 & 3 & \cdots \\
\text{Exponents} \\
5 & 4 & \cdots \\
\end{array} \]

shared memory

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

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

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

\[ \vdots \]

\[ \vdots \]

\[ x_1^{d-1} \ x_2^{d-1} \ x_3^{d-1} \ \cdots \ x_n^{d-1} \]
memory locations
we illustrate the work done by one thread

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 )</td>
<td>( x_1 )</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 )</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 )</td>
<td>( x_4 )</td>
</tr>
<tr>
<td>( x_2 x_3 x_4 )</td>
<td>( x_1 x_3 x_4 )</td>
<td>( (x_1 x_2) )</td>
<td>( x_1 x_2 x_3 )</td>
<td>( x_4 )</td>
</tr>
<tr>
<td>( \frac{\partial s}{\partial x_1} )</td>
<td>( \frac{\partial s}{\partial x_2} )</td>
<td>( \frac{\partial s}{\partial x_3} )</td>
<td>( \frac{\partial s}{\partial x_4} )</td>
<td>( (x_4 x_3) )</td>
</tr>
</tbody>
</table>

Only explicitly performed multiplications are marked by a star \( * \).
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 $\gamma = \frac{1}{c} \beta = x_1^3 x_2^7 x_3^4 x_4^5$.

<table>
<thead>
<tr>
<th>$L_1$</th>
<th>$L_2$</th>
<th>$L_3$</th>
<th>$L_4$</th>
<th>$L_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{\partial s}{\partial x_1} \star \alpha$</td>
<td>$\frac{\partial s}{\partial x_2} \star \alpha$</td>
<td>$\frac{\partial s}{\partial x_3} \star \alpha$</td>
<td>$\frac{\partial s}{\partial x_4} \star \alpha$</td>
<td>$\frac{\partial s}{\partial x_1} \star (3c)$</td>
</tr>
<tr>
<td>$\frac{\partial \gamma}{\partial x_1}$</td>
<td>$\frac{\partial \gamma}{\partial x_2}$</td>
<td>$\frac{\partial \gamma}{\partial x_3}$</td>
<td>$\frac{\partial \gamma}{\partial x_4}$</td>
<td>$\frac{\partial \gamma}{\partial x_1}$</td>
</tr>
<tr>
<td>$\frac{1}{3} \frac{\partial \gamma}{\partial x_1}$</td>
<td>$\frac{1}{7} \frac{\partial \gamma}{\partial x_2}$</td>
<td>$\frac{1}{4} \frac{\partial \gamma}{\partial x_3}$</td>
<td>$\frac{1}{5} \frac{\partial \gamma}{\partial x_4}$</td>
<td>$\frac{1}{3} \frac{\partial \gamma}{\partial x_1}$</td>
</tr>
</tbody>
</table>

Note that the coefficients $(3c), (7c), (4c), (5c)$ are precomputed. Only explicitly performed multiplications are marked by a star $\star$. 
regularity assumptions on the input data

Graphics compute processors exploit data parallelism.
Every thread evaluates and differentiates one monomial.

- On the one hand, to keep all 14 multiprocessors occupied about 1,000 monomials are needed.
- On the other hand, as monomials are stored as positions and exponents in constant memory, the 65,536 bytes of constant memory impose an upper bound on the number of monomials.

Let $n$ be the number of polynomials in the system, $m$ be the number of monomials per polynomial, $k$ be the number of variables per monomial, using one byte for a position and one byte for an exponent, then we need $n \times m \times k \times 2$ bytes.

As examples, we take $n = m$ between 30 and 40, and $k = n/2$. 
limits of shared memory capacity

With double double precision coefficients, dimension 70 is okay.

1. \((n/2 + 1) \times 2 \times \text{sizeof(double double)} \leq (70/2 + 1) \times 2 \times 16 = 1,152\) bytes in shared memory. To handle 32 monomials by a block of 32 threads we would need then at most

\[
32 \times 1,152 = 36,864 \text{ bytes of shared memory}.
\]

2. For storing values of the variable we would need

\[
n \times \text{sizeof(complex double double)} \leq 70 \times 2 \times \text{sizeof(double double)} = 70 \times 2 \times 16 = 2,240.
\]

3. Allocation both spaces in shared memory leaves

\[
(49,152 - (36,864 + 2,240)) > 10,000 \text{ bytes of shared memory}.
\]
computational experiments

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.