Hardware Acceleration Technologies in Computer Algebra: Challenges and Impact

Ph.D Thesis Presentation
of
Sardar Anisul Haque
Supervisor: Dr. Marc Moreno Maza

Ontario Research Centre for Computer Algebra
University of Western Ontario, London, Ontario

November 15, 2013

Overview and challenges

Overview

- This thesis deals with the implementation (both parallel and serial) of basic routines in computer algebra.
- These routines are from linear algebra (both sparse and dense) and polynomial arithmetic.
- We are interested in developing tools for analyzing algorithms and implementation techniques targeting hardware acceleration technologies.
- An outcome of our work is GPU-support for high-level polynomial system solver in the computer algebra system MAPLE.

Challenges

With respect to high-performance computing challenges, computer algebra low-level routines fall into categories:

(P1) memory access patterns are highly irregular and work count is essentially proportional to memory accesses;

(P2) the amount of work is much larger than the amount of reads/writes while memory access patterns are rather regular.

This classification defines the two parts of this thesis.

Two types problems of our interest in this thesis

Problems of type (P1)

- Irregular memory access patterns and relatively little work w.r.t. memory accesses
- Typical examples: sparse matrix and sparse polynomial arithmetic.
- Cases of interest involve huge data.
- All these properties make these operations not so suitable for manycores.
- Typical operations: sparse matrix vector multiplication (SpMxV) multiplication (Pinar and Heath, 1999). Solutions are classified broadly into two groups:
  - Improving locality by exploiting the structure of the data: by blocking (Vuduc and Moon, 2005) or rearranging data (Pinar and Heath, 1999).
  - Reducing the number of I/O operations in the whole memory hierarchy (Bender, Brodal, Fagerberg, Jacob, and Vicari 2010).

Problems of type (P2)

- Compute- and data-intensive operations, with regular memory access patterns
- Typical examples: dense matrix arithmetic and dense polynomial arithmetic.
- Fine grain parallelism; moreover certain complex memory access patterns (FFT) make these operations not so suitable for multicores.
- Typical operations: FFT-based and plain polynomial arithmetic. See the landmark book (von zur Gathen and Gerhard, 1999).
- Reports on GPU implementation: (Emeliyanenko, 2009-2011) (Morneo Maza & Pan, 2010-2011).

Plan

1. Around Sparse Matrix Vector Multiplication
   - Ideal Cache Model and Cache Complexity
   - Cache Friendly Sparse Matrix Vector Multiplication
   - One More Sorting Problem
   - Cache Oblivious Counting Sort Algorithm

2. A model of compuation for many-core architectures
   - Manycore architectures
   - Determinant on GPU by Condensation Method
   - Many-core Machine Model
   - Polynomial division algorithms
   - The Euclidean algorithm for polynomials
   - Polynomial multiplication algorithms
   - On the implementation and application of subproduct tree based technique
   - Integrating GPU support into bivariate solver

Plan

1. Around Sparse Matrix Vector Multiplication
   - Ideal Cache Model and Cache Complexity
   - Cache Friendly Sparse Matrix Vector Multiplication
   - One More Sorting Problem
   - Cache Oblivious Counting Sort Algorithm

2. A model of compuation for many-core architectures
   - Manycore architectures
   - Determinant on GPU by Condensation Method
   - Many-core Machine Model
   - Polynomial division algorithms
   - The Euclidean algorithm for polynomials
   - Polynomial multiplication algorithms
   - On the implementation and application of subproduct tree based technique
   - Integrating GPU support into bivariate solver

The (Z, L)-ideal cache model

Frigo, Leiserson, Prokop and Ramachandran (1999) introduce:

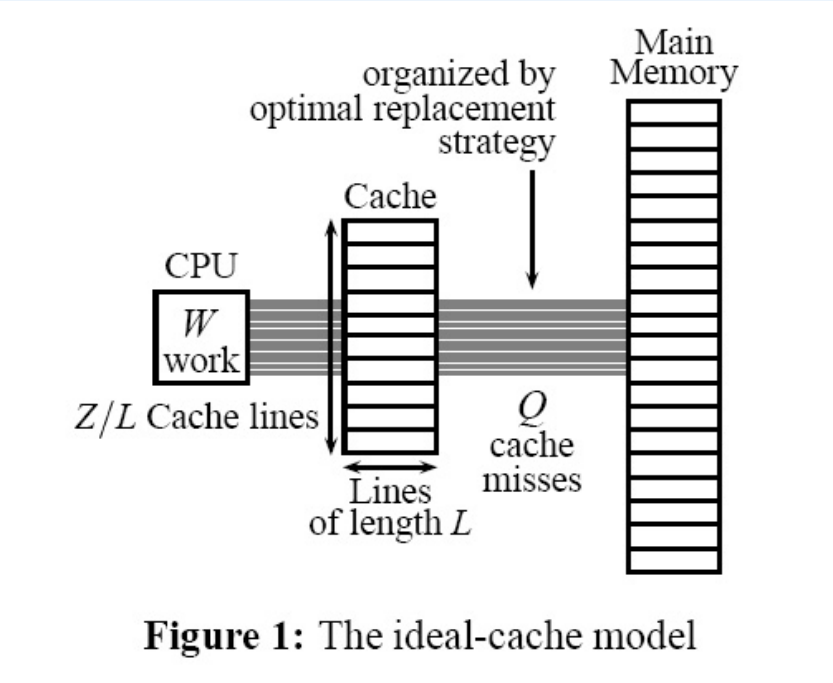


Figure 1: The ideal-cache model

- Computer with a **two-level memory hierarchy**:
  - an ideal (data) cache of Z words partitioned into Z/L *cache lines*, where L is the number of words per cache line.
  - an arbitrarily large main memory.
- Cache lines (sometimes called *blocks*) are the data unit when for transfer between cache and main memory.
- The cache is **tall**, that is, Z is much larger than L, say $Z \in \Omega(L^2)$.

Cache complexity

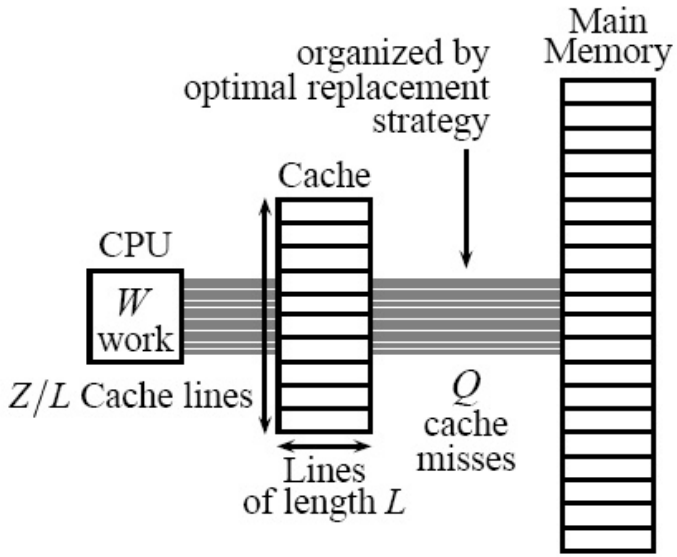


Figure 1: The ideal-cache model

- For an algorithm with input of size n, the ideal-cache model uses two complexity measures:
  - the **work** $W(n)$, which is its running time in the RAM model.
  - the **cache complexity** $Q(n; Z, L)$, that is, the number of cache misses the algorithm incurs (as a function of the size Z and line length L of the ideal cache).
- An algorithm is said to be **cache aware** if its behavior (and thus performances) can be tuned (and thus depend on) on the memory parameters (Z, L, etc.) of the targeted machine.
- Otherwise the algorithm is **cache oblivious**.

Cache complexity of SpMxV multiplication: an illustrative example (1/2)

Input matrix and vector

$$\overset{A}{\begin{pmatrix} a_{0,0} & 0 & 0 & 0 & a_{0,4} & 0 \\ 0 & 0 & a_{1,2} & 0 & 0 & a_{1,5} \\ 0 & a_{2,1} & 0 & a_{2,3} & 0 & 0 \end{pmatrix}} \times \overset{x}{\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}}$$

Cache misses due to x

Assume that the cache has 2 lines each of 2 words. Assume also that the cache is dedicated to store the entries from x:

$$\begin{bmatrix} & \end{bmatrix} \begin{bmatrix} x_0 & x_1 \\ & \end{bmatrix} \begin{bmatrix} x_0 & x_1 \\ x_4 & x_5 \end{bmatrix} \begin{bmatrix} x_0 & x_1 \\ x_2 & x_3 \end{bmatrix} \begin{bmatrix} x_4 & x_5 \\ x_2 & x_3 \end{bmatrix} \begin{bmatrix} x_4 & x_5 \\ x_0 & x_1 \end{bmatrix} \begin{bmatrix} x_2 & x_3 \\ x_0 & x_1 \end{bmatrix}$$

Total number of cache misses: 6.

Cache complexity of SpMxV multiplication: an illustrative example (2/2)

After reordering the columns of A

$$\underset{A'}{\begin{pmatrix} a_{0,0} & a_{0,4} & 0 & 0 & 0 & 0 \\ 0 & 0 & a_{1,2} & a_{1,5} & 0 & 0 \\ 0 & 0 & 0 & 0 & a_{2,1} & a_{2,3} \end{pmatrix}} \times \underset{x'}{\begin{pmatrix} x_0 \\ x_4 \\ x_2 \\ x_5 \\ x_1 \\ x_3 \end{pmatrix}}$$

Cache misses due to x'

Assume that the cache has 2 lines each of 2 words. Assume also that the cache is dedicated to store the entries from x:

$$\begin{bmatrix} & \end{bmatrix} \begin{bmatrix} x_0 & x_4 \\ & \end{bmatrix} \begin{bmatrix} x_0 & x_4 \\ & \end{bmatrix} \begin{bmatrix} x_0 & x_4 \\ x_2 & x_5 \end{bmatrix} \begin{bmatrix} x_0 & x_4 \\ x_2 & x_5 \end{bmatrix} \begin{bmatrix} x_0 & x_4 \\ x_1 & x_3 \end{bmatrix} \begin{bmatrix} x_0 & x_4 \\ x_1 & x_3 \end{bmatrix}$$

Total number of cache misses: 3.

Gray codes

Definition

- For $N = 2^n$, an n-bit code $C_n = (u_1, u_2, \ldots, u_N)$, where $N = 2^n$, is a *Gray code* if u_i and u_{i+1} differ in exactly one bit, for all i.
- This corresponds to a Hamiltonian path (cycle) in the n-dimensional hypercube.

Gray codes

Definition

- For $N = 2^n$, an n-bit code $C_n = (u_1, u_2, \ldots, u_N)$, where $N = 2^n$, is a *Gray code* if u_i and u_{i+1} differ in exactly one bit, for all i.
- This corresponds to a Hamiltonian path (cycle) in the n-dimensional hypercube.

Binary reflected Gray codes

The reflected Gray code Γ^n is defined recursively by

$$\Gamma^1 = (0, 1) \quad \text{and} \quad \Gamma^{n+1} = \mathbf{0}\,\Gamma^n,\ \mathbf{1}\,\Gamma^{n\mathrm{R}}$$

Introduced by Frank Gray 1953 for shaft encoders

$$\Gamma^3 = 000, 001, 011, 010, 110, 111, 101, 100.$$

Gray codes

Definition

- For $N = 2^n$, an n-bit code $C_n = (u_1, u_2, \ldots, u_N)$, where $N = 2^n$, is a *Gray code* if u_i and u_{i+1} differ in exactly one bit, for all i.
- This corresponds to a Hamiltonian path (cycle) in the n-dimensional hypercube.

Binary reflected Gray codes

The reflected Gray code Γ^n is defined recursively by

$$\Gamma^1 = (0, 1) \quad \text{and} \quad \Gamma^{n+1} = \mathbf{0}\,\Gamma^n,\ \mathbf{1}\,\Gamma^{n\mathrm{R}}$$

Introduced by Frank Gray 1953 for shaft encoders

$$\Gamma^3 = 000, 001, 011, 010, 110, 111, 101, 100.$$

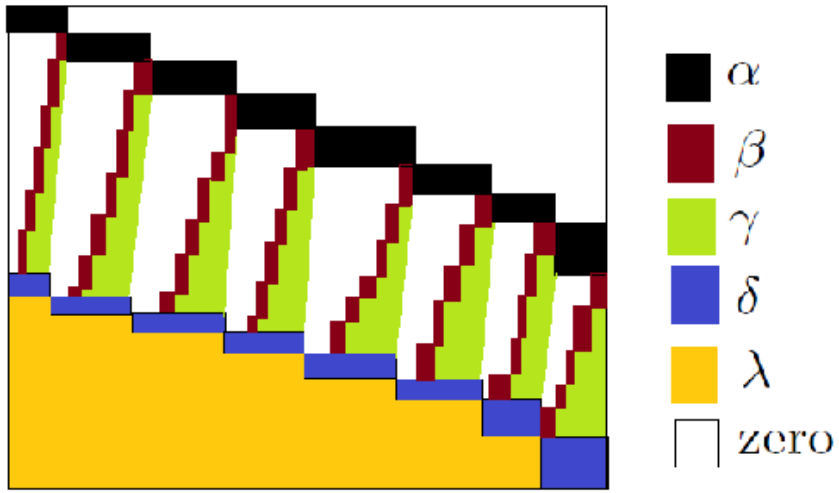
Binary reflected Gray code for arithmetic operations

- Integers of dimension m can be represented by a data structure that uses $m + \log m + O(\log\log m)$ bits so that increment and decrement operations require at most $\log m + O(\log\log m)$ bit inspections and 6 bit changes per operation. (M.Z. Rahman and J.I. Munro, 2007).
- They have also good results for addition and subtraction.

Cache friendly SpMxV multiplication: our method

Principle

For a sparse matrix A with m rows and n columns, we re-order columns (and rows) so as to reduce cache complexity when multiplying A by a dense vector.



Details

1. Consider each column as a binary string from the binary reflected Gray code Γ^m.
2. We partition columns based on their rank, considering only a few most significant nonzeros.
3. Next, we re-order rows with no significant nonzeros according to Γ^n.
4. Keep Refining the column partition (by considering more nonzeros) until each part of the partition is a singleton.

Time complexity

In theory . . .

- Recall that A has n columns and m rows.
- Theoretically, one could sort the columns of A according to their rank in Γ^m. The algebraic complexity of this preprocessing would be $O(n \log(n) f(m))$ where $f(m)$ is the maximum number of bit operations for comparing two columns.
- Using sparse data-structure for encoding A, we can assume that $f(m)$ is bounded over by the maximum number of nonzeros in a column.
- Let τ be the total number of non-zeros in A.
- This "direct" approach would requires at most $O(\tau \log(n))$ bit operations.

Time complexity

In theory . . .

- Recall that A has n columns and m rows.
- Theoretically, one could sort the columns of A according to their rank in Γ^m. The algebraic complexity of this preprocessing would be $O(n \log(n) f(m))$ where $f(m)$ is the maximum number of bit operations for comparing two columns.
- Using sparse data-structure for encoding A, we can assume that $f(m)$ is bounded over by the maximum number of nonzeros in a column.
- Let τ be the total number of non-zeros in A.
- This "direct" approach would requires at most $O(\tau \log(n))$ bit operations.

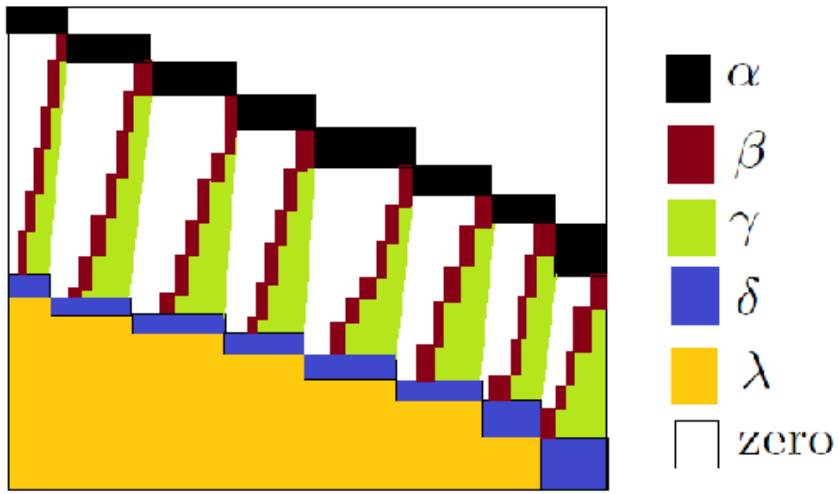
Our results

- Our preprocessing procedures requires $O(\tau)$ index comparisons and $O(\tau + n)$ data-structure updates,.
- Experimentally, we verified that our preprocessing is faster than procedures based on sorting algorithms
- Typically, for $10^6 \leq m, n \leq 10^8$ and $n \leq \tau \leq 10n$, preprocessing cost is less than 70 SpMxV
- Therefore, preprocessing cost can be amortized in iterative methods like the conjugate gradient algorithm, where the number of iterations is $O(n)$, often $\Theta(n)$.

Cache complexity

Our results

- Recall that nonzeros are classified into five types: α, β, γ, δ and λ.
- Proposition: if $\tau \geq 2n$ then the total number of nonzeros of types α or β is at least $2n$.
- Experimentally, the total number of nonzeros of types α, β and δ is at least $3n$.
- Theorem: Assuming $Z \geq 2\sqrt{nL}$, the number of cache misses incured when multiplying the nonzeros of types α, β and δ is at most the $3n/L + O(\sqrt{n/L})$.



One more sorting problem

Sorting long binary strings

- Kirkpatrick and Reisch (1984) ask the following question: "For what ranges of inputs can we construct practical $o(n \log n)$ integer sorting algorithms?"
- Comparison-based sorting requires $n \log(n) f(m)$, where m is the maximum bit-length of a string and $f(m)$ is the maximum number of bit operations for comparing two strings of size at most m.
- Our proposed preprocessing step for SpMxV multiplication suggests an algorithm for sorting many long strings, in particular when those latter are *sparse*.

Our algorithm for sorting long integers

The input integers are encoded by their binary expansions, all assumed to be of equal length.

1. We partition these integers based on the index of their Most Significant Bit (MSB).
2. We refine all parts of the first partition all together as follows:
   - we sort all integers by sorting the indices of their Second Most Significant Bit by means of a counting sort (which is a stable sort) running in expectation within $O(n+p)$ bit operations, where $1/p$ is the probability that a random bit in an integer is 1. Note that: $p = mn/\tau$.
   - then we retrieve the part of each integer in $O(n)$ bit operations by means of carefully designed data-structures.
3. We keep refining partition considering 3rd, 4th, etc Most Significant Bit until each part of the partition is a singleton.

Our sorting algorithm: an illustrative example

Let a, b, c, d, e, f be given integers

a	b	c	d	e	f
0	0	1	0	1	0
1	1	0	1	0	0
0	0	0	0	1	1
1	1	0	0	0	0
0	0	1	1	0	0

Running the algorithm

Each integer is encoded by the indixes of its 1s:

$$a = (1,3), b = (1,3), c = (0,4), d = (1,4), e = (0,2), f = (2)$$

Initial partition

$$= ((a, b, c, d, e, f)).$$

Sort the integers based on their MSBs leads to

$$= ((c, e), (a, b, d), (f)).$$

Sort the integers based on their second MSB leads to

$$((c), (e), (a, b), (d), (f)).$$

Complexity analysis of our sorting algorithm

Theorem

- Recall that we are sorting n integers of bit-size m forming a bit-matrix A with τ nonzero bits
- Recall that $p = mn/\tau$ is the average distance between two consecutive nonzero bits in a column.
- Assume that A is sparse in the sense that $O(\log_p(n)) \in \Theta(1)$.
- Then, the expected number of iterations of our sorting algorithm is $O(\log_p(n))$.
- Moreover it is expected to run within $O(n + p + n\frac{p}{2(p-1)})$ bit operations.

Characterisitics of the test matrices

Matrix name	m	n	τ
fome21	67748	216350	465294
lp_ken_18	105127	154699	358171
barrier2-10	115625	115625	3897557
rajat23	110355	110355	556938
hcircuit	105676	105676	513072
GL7d24	21074	105054	593892
GL7d17	1548650	955128	25978098
GL7d19	1911130	1955309	37322725
wikipedia-20051105	1634989	1634989	19753078
wikipedia-20070206	3566907	3566907	45030389

Table: Test matrices from University of Florida Sparse Matrix Collections.

SpMxV multiplication time for test matrices

Experimental results

Matrix name	With our preprocessing	No preprocessing	Random re-ordering
fome21	3.6	3.9	4.8
lp_ken_18	2.7	3.1	3.3
barrier2-10	19.0	19.1	23.2
rajat23	3.0	3.0	3.4
hcircuit	2.6	2.5	2.9
GL7d24	3.0	3.2	3.1
GL7d17	484.6	625.0	580.7
GL7d19	784.6	799.0	899.2
wikipedia-20051105	258.9	321.0	411.5
wikipedia-20070206	731.5	859.0	1046.0

Table: CPU time in seconds for 1000 *SpMxVs*.

Cache oblivious counting sort algorithm

Counting sort and locality issues

- The classical *Counting Sort* runs in linear time w.r.t. number of entries and their range size.
- Unfortunately, *Counting Sort* suffers from poor data locality due to **random memory accesses**. Some comparison-based sorting implementation outperform it.
- Nevertheless *Counting Sort* should work well if the range of integers fits in cache.

An improvement

- Sorting an input array A of n integers in the range $[0, r]$ by counting sort incures at most $\mathbf{3n} + 2n/L + 2r/L + 4$ cache misses.
- Assume $r = \ell\, m - 1$ for non-negative integers ℓ and m are such that $m < Z/(1+L)$.
- We say that A is *m*-buckteted whenever, for all $j = 0 \cdots (\ell - 1)$, every entry of A lying in the sub-range $[jm, (j+1)m - 1]$ appears in A before every entry of A lying in the sub-range $[(j+1)m, \ell m - 1]$.
- To improve data locality, we preprocess A such that it becomes *m*-buckteted.
- Letting $u = \log_m(r) - 1$, the number of cache misses to preprocess and counting-sorting A is

$$Q = 3n(u+1)/L + mu/L + u + (2+m)\left(\frac{m^u - 1}{m-1} + \ell\right) + 4r/L + 4.$$

If $u = 1$, that is, $r = m^2$, then Q simplifies to $6n/L + 2m + m/L + r + 4r/L + 5$.

Experimental results

n	classical counting sort	cache-oblivious counting sort (preprocessing + sorting)
100000000	13.74	4.66 (3.04 + 1.62)
200000000	30.20	9.93 (6.16 + 3.77)
300000000	50.19	16.02 (9.32 + 6.70)
400000000	71.55	22.13 (12.50 +9.63)
500000000	94.32	28.37 (15.71 + 12.66)
600000000	116.74	34.61 (18.95 + 15.66)

Table: CPU times in seconds for both classical and cache-oblivious counting sort algorithm.

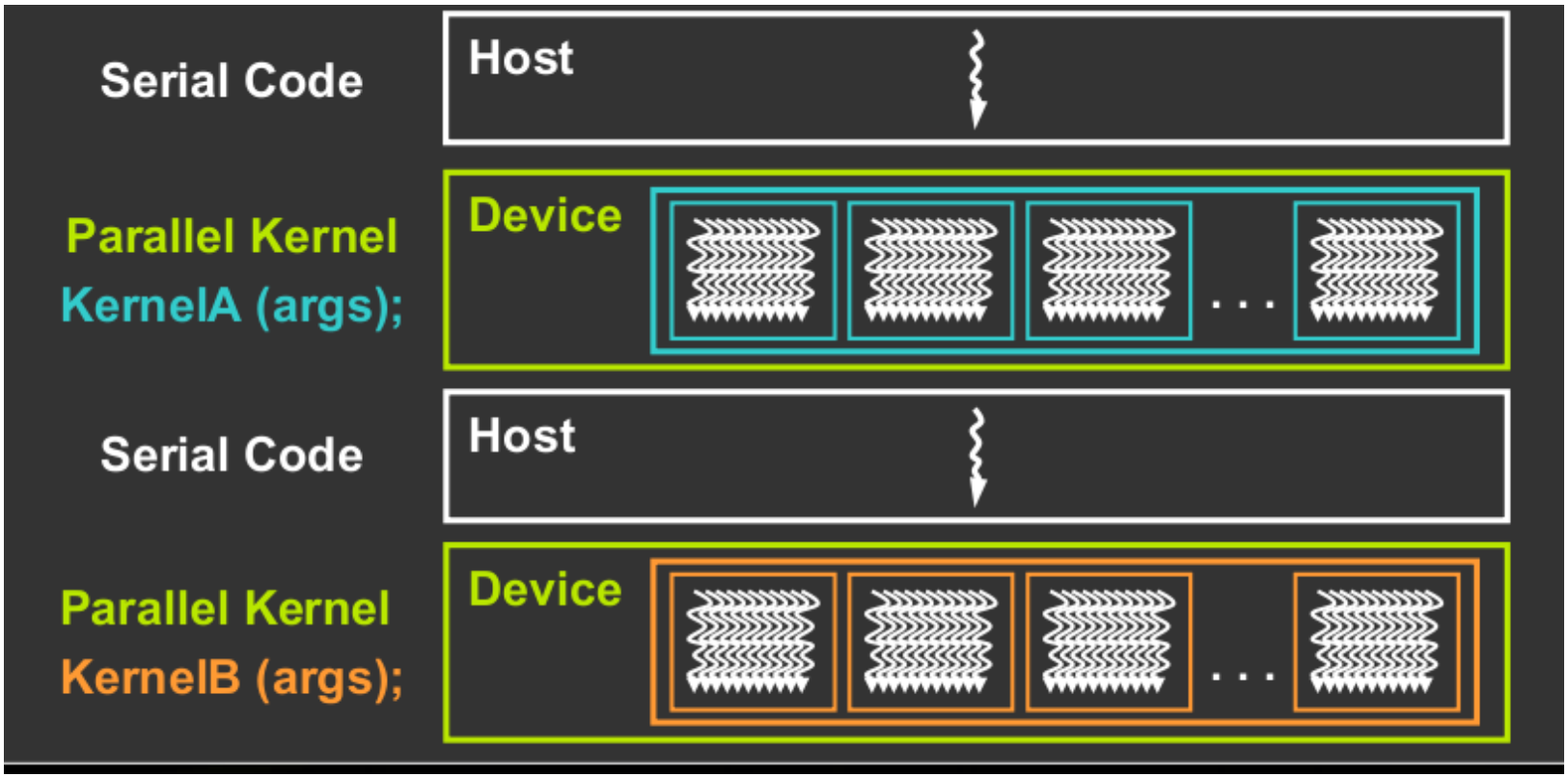
Plan

1. Around Sparse Matrix Vector Multiplication
   - Ideal Cache Model and Cache Complexity
   - Cache Friendly Sparse Matrix Vector Multiplication
   - One More Sorting Problem
   - Cache Oblivious Counting Sort Algorithm

2. A model of compuation for many-core architectures
   - Manycore architectures
   - Determinant on GPU by Condensation Method
   - Many-core Machine Model
   - Polynomial division algorithms
   - The Euclidean algorithm for polynomials
   - Polynomial multiplication algorithms
   - On the implementation and application of subproduct tree based technique
   - Integrating GPU support into bivariate solver

Manycore programming (1/2)

- The parallel kernel C code executes in many device threads across multiple GPU processing elements, called streaming processors (SP).
- Thus, the parallel code (kernel) is launched and executed on a device by many threads.
- Threads are grouped into thread blocks.
- One kernel is executed at a time on the device.
- Many threads execute each kernel.
- Thus, each thread executes the same code on different data based on its thread and block ID.



Manycore programming (2/2)

Challenges:

- data structures and algorithm design are not the same as in conventional multicore programming.
- data transfers between host memory and device main memory increase overhead.
- data transfers within the device memory hierarchy should be carefully designed.
- synchronization among threads of different thread block increase overhead (may need redesign the algorithm).
- **Little software support to help with code development.**
- **Lack of theoretical models supporting code development.**

Computing determinant on GPU by condensation method (1/3)

Overview

- Charles Lutwidge Dodgson (1866) developed the condensation method for computing determinant of a square matrix $A = (a_{i,j} \mid 0 \leq i, j \leq n-1)$ of order n
- Salem and Said (2007) tuned into a complete algorithm: Let ℓ be the smallest column index of a non-zero element in the first row of A. The *condensation step* produces a matrix $B = (b_{i,j})$ of order $n-1$ defined by:

$$b_{i,j} = \begin{vmatrix} a_{0,\ell} & a_{0,j+1} \\ a_{i+1,\ell} & a_{i+1,j+1} \end{vmatrix}$$

for $j \geq \ell$ and by $b_{i,j} = -a_{i+1,j}a_{0,\ell}$ for $j < \ell$. The key relation between A and B is the following:

$$\det(A) = \det(B)/(a_{0,\ell})^{n-2}$$

- We designed a GPU implementation of Salem and Said's algorithm.

Experimental results with modular integers

We parallelize condensation method on GPU.

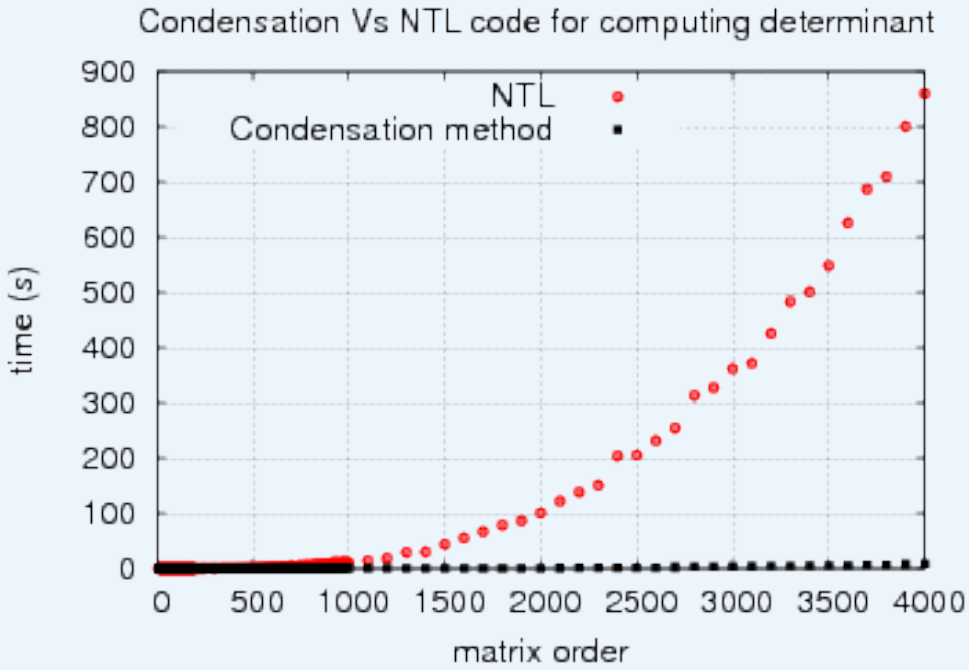


Figure: CUDA code for condensation method and determinant on NTL over finite field.

Computing determinant on GPU by condensation method (3/3)

Experimental results with floating point coefficients

Matrix order	Maple	MATLAB	Condensation Method on GPU
5	0.3239712e-11	3.749295e-12	3.74967e-12
6	-0.1037653175e-16	5.367300e-18	5.36556e-18
7	-0.2940657217e-22	4.835803e-25	4.44292e-25
8	-0.2156380381e-28	2.737050e-33	-3.92813e-33
9	-0.1692148341e-35	9.720265e-43	-2.79235e-41
10	0.4704819751e-42	2.164405e-53	-4.44342e-50
15	0.1386122551e-74	-2.190300e-120	-9.47742e-103
20	0.4711757502e-106	-1.100433e-195	3.81829e-164
25	-0.4092672466-139	5.482309e-274	-3.82134e-239
30	-0.2087134536-174	0	-2.50914e-319
35	0.6863051439e-205	-	3.50293e-398
40	0.3354475665e-237	-	-7.42227e-479

Table: Determinant of Hilbert Matrix by `Maple`, MATLAB, and condensation method on both CPU and GPU.

Models of computations targeting many-core architectures

Popular models

- PRAM (parallel random access machine) supports data parallelism but not task parallelism. Moreover, cannot support memory traffic issues (cache complexity, memory contention)
- Queue Read Queue Write PRAM considers memory contention, however, it unifies in a single quantity time spent in arithmetic operations and time spent in read/write accesses
- TMM (Threaded Many-core Memory) model retains many important characteristics of GPU-type architectures, however, the running time estimate on P cores is not given by a Graham-Brent theorem

In this work:

We propose a many-core machine model (MMM) which aims at optimizing algorithms targeting implementation on GPUs. We insist on

- Two-level DAG (directed acyclic graph) programs
- Parallelism overhead
- A Graham-Brent theorem

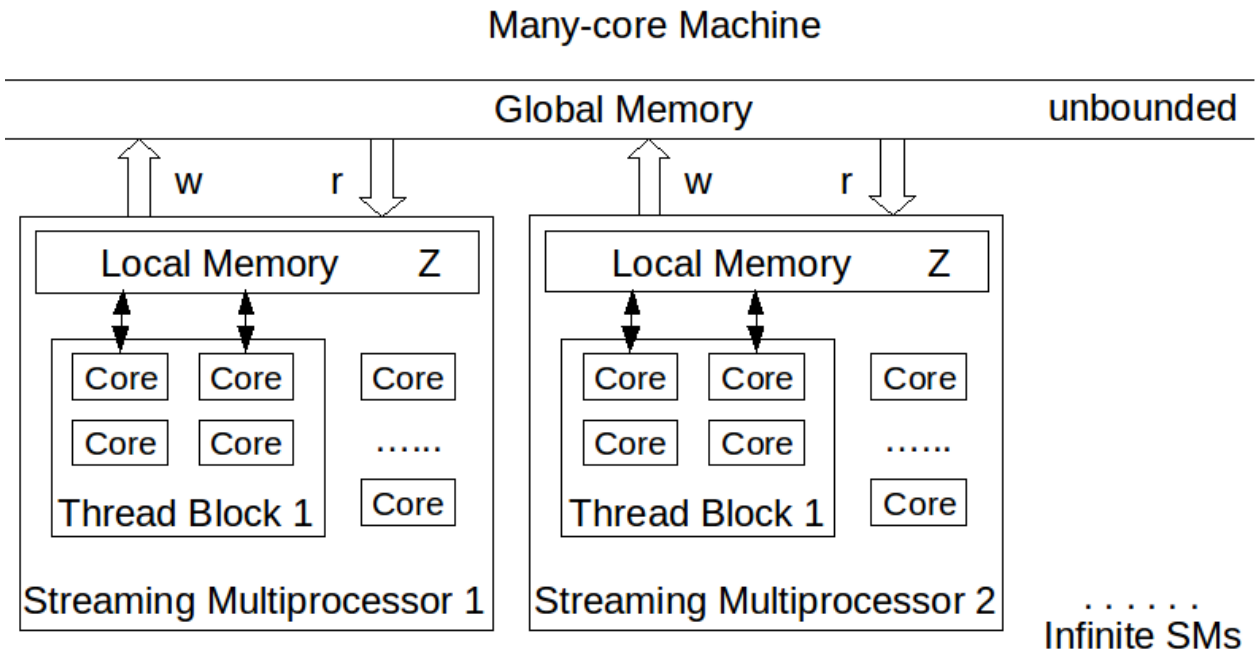
Motivations

- At HPCS 2012, we reported on an optimized GPU implementation of polynomial arithmetic operations (division, GCD, multiplication)
- These optimizations were obtained by minimizing data transfer between global and local memories and also by minimizing the impact of code divergence in kernels

Using the MMM for minimizing parallelism overheads

- Let s be a program parameter of an MMM program P that can be arbitrarily chosen in some range $\mathcal{S}$. Let s_0 be a particular value of s.
- Assume that, when s varies in $\mathcal{S}$, the work, say $W_{\mathcal{P}}(s)$, and the span, say $S_{\mathcal{P}}(s)$, do not vary much, that is, $W_{\mathcal{P}}(s_0)/W_{\mathcal{P}}(s) \in \Theta(1)$ and $S_{\mathcal{P}}(s_0)/S_{\mathcal{P}}(s) \in \Theta(1)$ hold.
- Assume also that the parallelism overhead $O_{\mathcal{P}}(s)$ varies more substantially, say $O_{\mathcal{P}}(s_0)/O_{\mathcal{P}}(s) \in \Theta(|s - s_0|)$.
- Then, we determine a value $s_{\min} \in \mathcal{S}$ which maximizes the ratio $O_{\mathcal{P}}(s_0)/O_{\mathcal{P}}(s)$.
- We use our version of Graham-Brent's theorem to check that the upper bound for the running time (on P streaming multiprocessors) of $\mathcal{P}(s_{\min})$ is no more than that of $\mathcal{P}(s_o)$.

MMM: characteristics

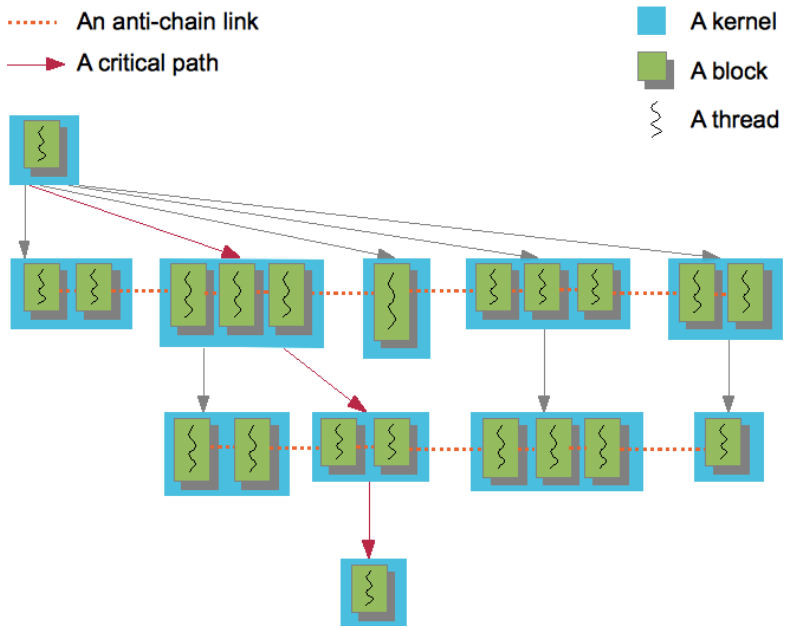


Architecture:

- Unbounded number of *streaming multiprocessors* (SMs) which are all identical
- Each SM has a finite number of processing cores and a fixed-size local memory
- 2-level memory hierarchy, comprising an unbounded global memory with high latency and low throughput while the SM local memories have low latency and high throughput

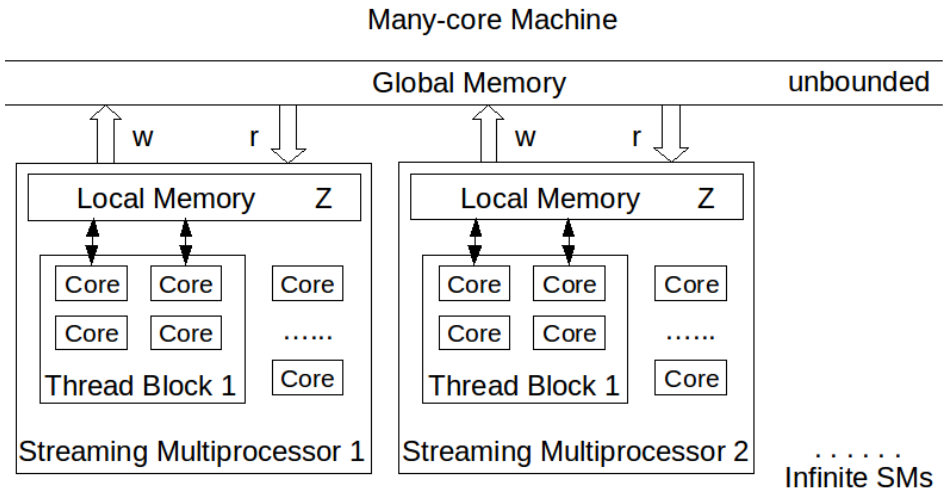
MMM: programs

Each MMM program $\mathcal{P}$ is modeled by a directed acyclic graph $(\mathcal{K}, \mathcal{E})$, called the **kernel DAG** of $\mathcal{P}$, where each node $K \in \mathcal{K}$ represents a kernel, and each edge $E \in \mathcal{E}$ represents a kernel call which must precede another kernel call.



- Note: a kernel call can be executed whenever all its predecessors in the DAG $(\mathcal{K}, \mathcal{E})$ have completed their execution
- Since each kernel of the program $\mathcal{P}$ decomposes into a finite number of thread-blocks, we map $\mathcal{P}$ to a second graph, called the **thread block DAG** of $\mathcal{P}$, whose vertex set $\mathcal{B}(\mathcal{P})$ consists of all thread-blocks of the kernels of $\mathcal{P}$, such that (B_1, B_2) is an edge if B_1 is a thread-block of a kernel preceding the kernel of B_2 in P.

MMM: scheduling, synchronization and memory access policy



Scheduling and synchronization:

- At run time, an MMM machine schedules thread-blocks onto the SMs, based on the dependencies among kernels and the hardware resources required by each thread-block
- Threads within a thread-block cooperate with each other via the local memory
- Thread-blocks interact with each other via the global memory

Memory access policy:

- All threads of a given thread-block can access simultaneously any memory cell of the local memory or the global memory
- Read/Write conflicts are handled by the CREW (concurrent read exclusive write) policy

MMM: machine parameters

For the purpose of analyzing program performance, we define two *machine parameters*

- U: Time (expressed in clock cycles) to transfer one machine word between global memory and the local memory of any SM
- Z: Size (expressed in machine words) of the local memory of each SM

For a thread-block B, if each thread executes at most ℓ local (i.e. arithmetic) operations, and reads r (resp. writes w) words to the global memory, then to compute the total running time T of an SM executing B,

- the total time T_D spent in data transfer between the global memory and the local memory

$$T_D \leq (r + w)\, U$$

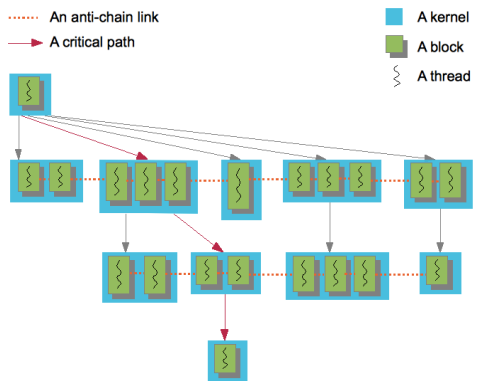
- there exists a constant V such that the total time T_A spent in local operations satisfies

$$T_A \leq \ell\, V$$

we have

$$T = T_A + T_D \leq \ell + (r + w)\, U, \text{ with } V = 1.$$

MMM: complexity measures



Work:

- The *work* $W(B)$ of a thread-block B is defined as the total number of local operations performed by the threads of B
- The *work* $W(K)$ of a kernel K is defined as the sum of the works of its thread-blocks
- The *work* $W(\mathcal{P})$ of the entire program $\mathcal{P}$ is defined as the total work of all its kernels

$$W(\mathcal{P}) = \sum_{K \in \mathcal{K}} W(K)$$

Parallelism overhead:

- The *overhead* $O(B)$ of a thread-block B is defined as $(r + w)\,U$, assuming that each thread of B reads (at most) r words and writes (at most) w words to the global memory
- The *overhead* $O(K)$ of a kernel K is defined as the sum of the overheads of its thread-blocks
- The *overhead* $O(\mathcal{P})$ of the entire program $\mathcal{P}$ is defined as the total overhead of all its kernels

$$O(\mathcal{P}) = \sum_{\alpha} O(\alpha)$$

MMM: complexity measures

Span:

- The *span* $S(B)$ of a thread-block B is defined as the maximum number of local operations performed by a thread of B
- The *span* $S(K)$ of a kernel K is defined as the maximum span of its thread-blocks
- We define the span $S(\gamma)$ of any path γ from the first kernel to a last one as

$$S(\gamma) = \sum_{K \in \gamma} S(K)$$

- The *span* $S(\mathcal{P})$ of the entire program $\mathcal{P}$ is defined as

$$S(\mathcal{P}) = \max_{\gamma} S(\gamma)$$

Theorem (Graham-Brent)

We have the following estimate for the running time T_p of the program $\mathcal{P}$ when executed on p SMs

$$T_p \leq (N(\mathcal{P})/p + L(\mathcal{P})) \cdot C(\mathcal{P}),$$

where

$N(\mathcal{P})$ *number of vertices in the thread-block DAG of $\mathcal{P}$,*

$L(\mathcal{P})$ *critical path length (that is, the length of the longest path) in the thread-block DAG of $\mathcal{P}$,*

$C(\mathcal{P})$ $= \max_{B \in \mathcal{B}(\mathcal{P})} (S(B) + O(B))$.

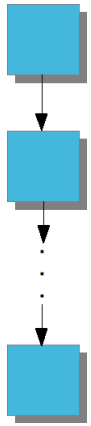
Plain division algorithms for polynomials

Given two polynomials a and b over a finite field $\mathbb{K}$ and with variable $\mathbf{X}$, where $\deg(a) = n-1$, and $\deg(b) = m-1$, compute the remainder in the Euclidean division of a by b.

- Naive division algorithm
- Optimized division algorithm

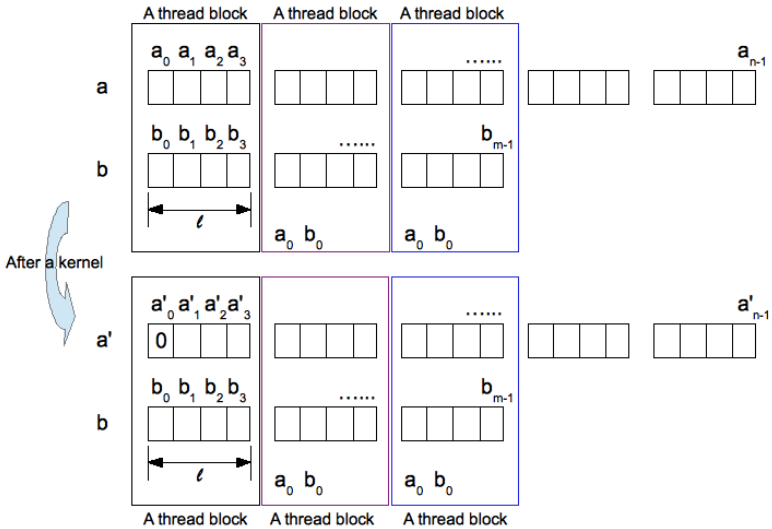
We assume that

- b is not zero
- $n \geq m$



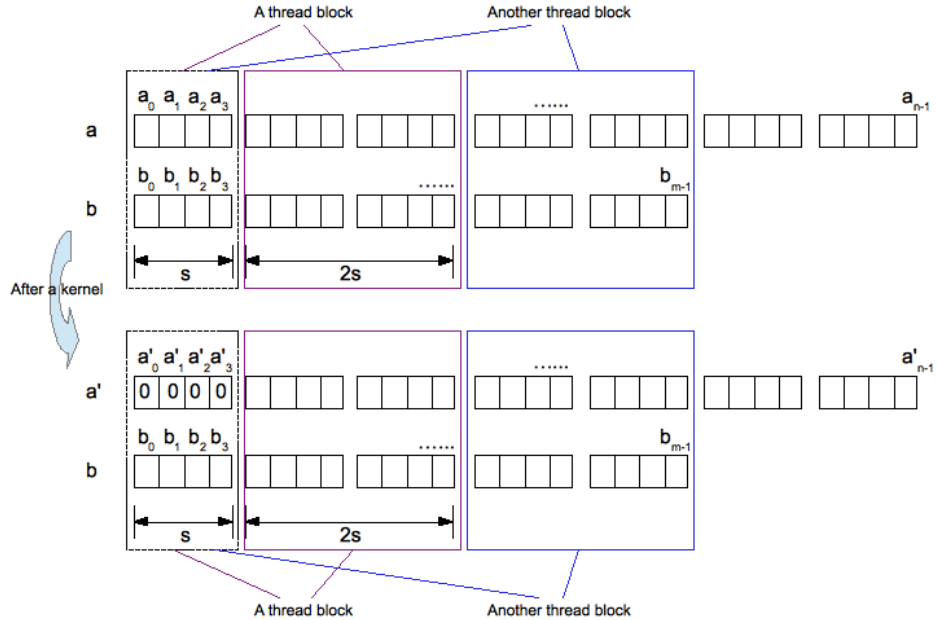
Plain division algorithms

Naive Division Algorithm



- Each kernel performs 1 division step
- $n - m + 1$ kernels are executed in serial

Optimized Division Algorithm



- Each kernel performs s division steps
- $\lceil \frac{n-m+1}{s} \rceil$ kernels are executed in serial

Analysis of plain division algorithms

We obtain the work ratio and the overhead ratio as

$$\frac{W_{\mathrm{nai}}}{W_{\mathrm{opt}}} = \frac{8\,(Z+1)}{9\,Z+7} \quad \text{and} \quad \frac{O_{\mathrm{nai}}}{O_{\mathrm{opt}}} = \frac{20}{441}\,Z$$

Applying Theorem 1,

$$R = \frac{(N_{\mathrm{nai}}/p + L_{\mathrm{nai}}) \cdot C_{\mathrm{nai}}}{(N_{\mathrm{opt}}/p + L_{\mathrm{opt}}) \cdot C_{\mathrm{opt}}} = \frac{2}{3}\,\frac{(3+5\,U)\,(2\,m+Z\,p)\,Z}{(Z+21\,U)\,(7\,m+2\,Z\,p)}$$

When m escapes to infinity, the ratio R is equivalent to

$$\frac{4}{21}\,\frac{(3+5\,U)\,Z}{Z+21\,U}$$

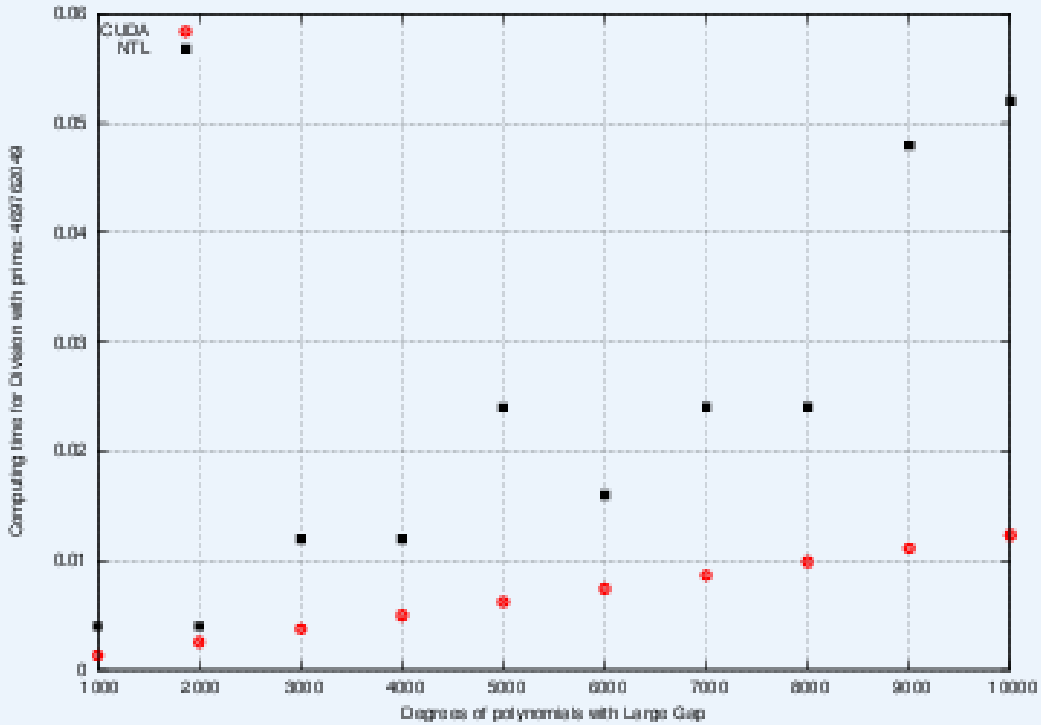
- We observe that this latter ratio is larger than 1 if and only if $Z > \frac{441\,U}{20\,U-9}$ holds
- The optimized algorithm is overall better than the naive one

Experimental results

Optimized Vs naive

Optimized division is almost 4 times faster than naive division.

Optimized Vs NTL library



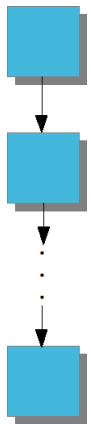
The Euclidean algorithm for polynomials

Given two polynomials a and b over a finite field $\mathbb{K}$ and with variable $\mathbf{X}$, where $\deg(a) = n-1$ and $\deg(b) = m-1$, compute the greatest common divisor (GCD) of a and b.

- Naive Euclidean algorithm
- Optimized Euclidean algorithm

We assume that

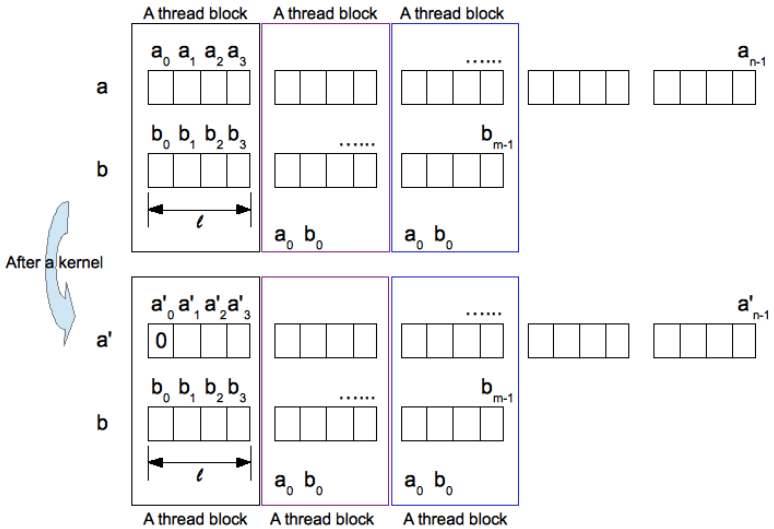
- b is not zero
- $n \geq m$



The Euclidean algorithm

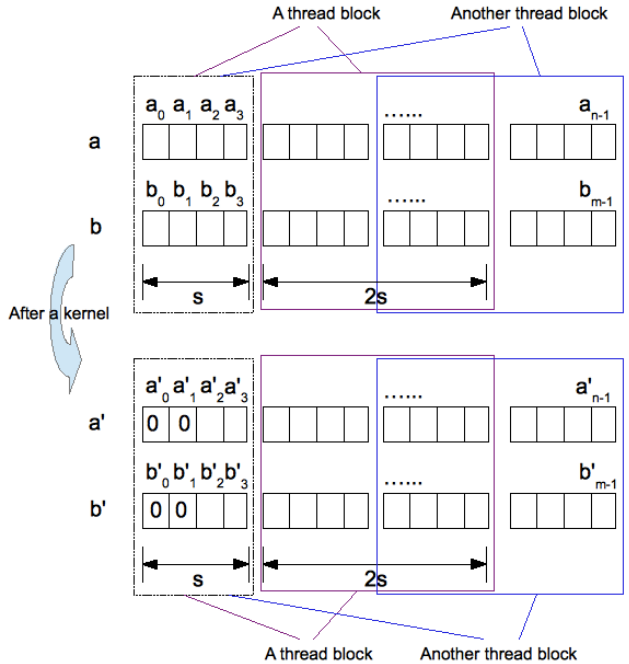
It checks the current degree of both a and b to decide which polynomial will take the role as a divisor, and then it completes a division step

Naive Euclidean algorithm



- Each kernel performs 1 division step
- $n + m - 2$ kernels are executed in serial

Optimized Euclidean algorithm



- Each kernel performs s division steps
- $\frac{n+m}{s}$ kernels are executed in serial

Analysis of the Euclidean algorithm

We obtain the work ratio and the overhead ratio, replacing m by n as

$$\frac{W_{\mathrm{nai}}}{W_{\mathrm{opt}}} = \frac{(284\,Z+2)\,n^2+(Z-2)\,n}{(1296\,Z+7488)\,n^2+(348\,Z^2+2208\,Z)\,n-(115\,Z^3+616\,Z^2)}$$

$$\frac{O_{\mathrm{nai}}}{O_{\mathrm{opt}}} = \frac{5}{48}\,\frac{Z(2\,n+2+Z)}{6\,n+Z}$$

- As n escapes to infinity, the additional work $W_{\mathrm{opt}} - W_{\mathrm{nai}}$ is only a portion of W_{nai},
- Meanwhile the data transfer overhead decreases as Z increases.

Applying Theorem 1, when n escapes to infinity, the ratio R is equivalent to

$$R = \frac{(N_{\mathrm{nai}}/p + L_{\mathrm{nai}})\cdot C_{\mathrm{nai}}}{(N_{\mathrm{opt}}/p + L_{\mathrm{opt}})\cdot C_{\mathrm{opt}}} \simeq \frac{(3+5\,U)\,Z}{9\,(Z+16\,U)}$$

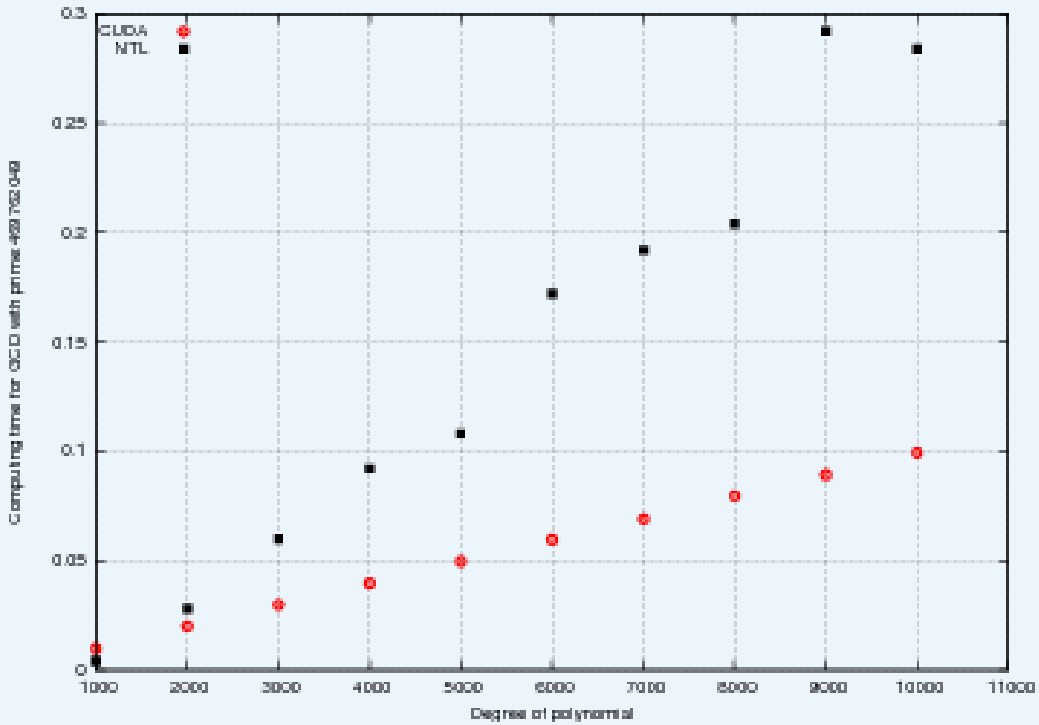
- We observe that this latter ratio is larger than 1 if and only if $Z > \frac{144\,U}{5\,U-6}$ holds
- The optimized algorithm is overall better than the naive one

Experimental results

Optimized Vs Naive

Optimized gcd is almost 4 times faster than naive gcd.

Optimized Vs NTL library



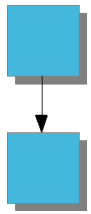
Analysis of (plain) polynomial multiplication

Given two polynomials a and b over a finite field $\mathbb{K}$ and with variable $\mathbf{X}$, where $\deg(a) = n-1$ and $\deg(b) = m-1$, compute the product d of $a \times b$

1) Multiplication phase
2) Addition phase

We assume that

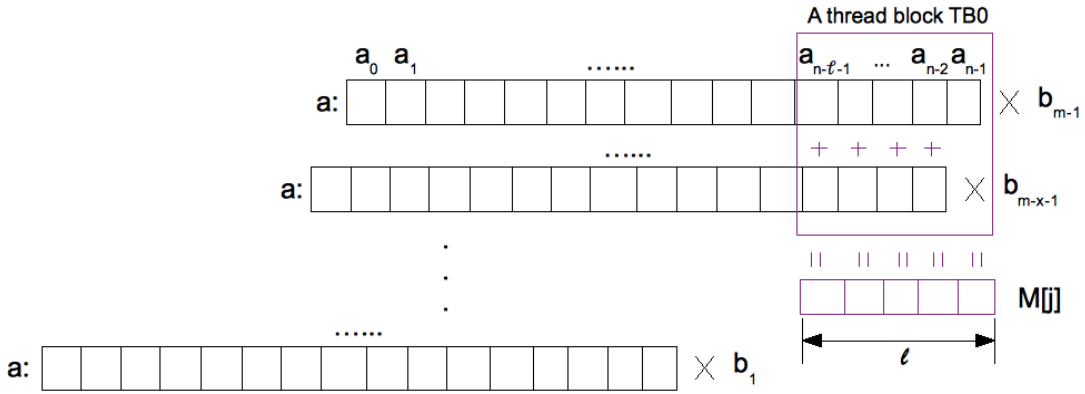
- $n \geq m$



Multiplication phase

Algorithm 1: MulKer(a, b, M, n, x)

Input: $a, b, M \in \mathbb{K}[X]$ and an integer $x \geq 1$.
j = blockID×blockDim+threadID; t = threadID;
Let B' and A' be two local arrays of size x and blockDim respectively with coefficients in $\mathbb{K}$;
$i = j \mod (n + x - 1)$;
if $i \geq n$ **then**
  $A'[t] = 0$;
else
  $A'[t] = a[i]$;
if $t < x$ **then**
  $B'[t] = b[\lfloor (j/(n + x - 1)) \rfloor x + t]$;
`/* copying from global */`
$f = 0$;
for $(k = 0; k < x \wedge (i - k) \geq 0; k = k + 1)$ **do**
  $f = f + A'[i - k]\, B'[k]$;
$M[j] = f$;
`/* writing to global memory */`

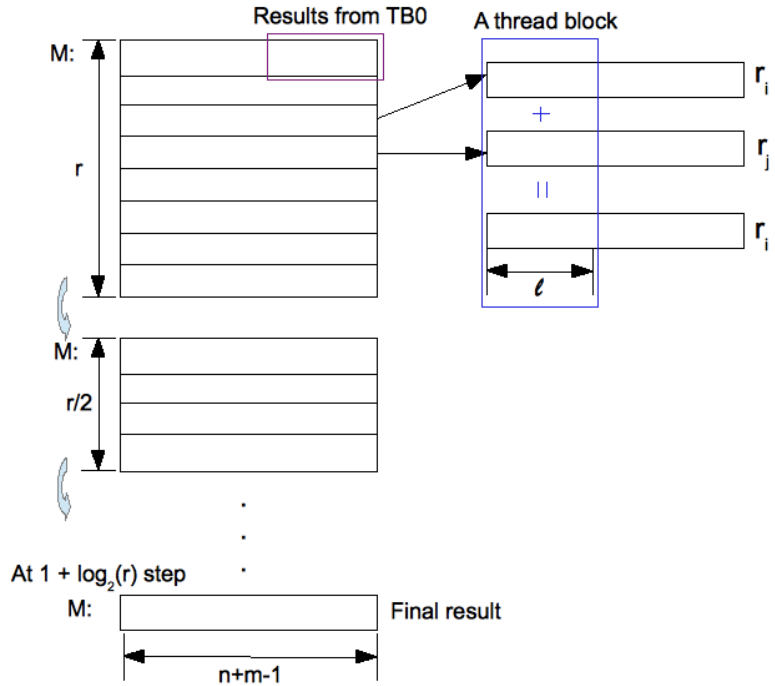


- Within a ℓ-thread block, (1) each thread reads a coefficient from a, (2) x threads read a coefficient from b and (3) ℓ partial sums are written to M

Addition phase

Algorithm 2: AddKer(M, d, c, x, r, i)

Input: $M, d, \in \mathbb{K}[X]$ and c, x, r, i are positive integers.
j =blockID blockDim +threadID; t =threadID;
$k = j \mod c$;
$q = \lfloor j/c \rfloor$;
$s = 2^i - 1 + 2^{i+1} q$;
$e = s + 2^i$;
if $k < 2^i x$ **then**
  $d[s x + k] = d[s x + k] + M[s r + k]$;
else
  $M[e r + k - 2^i x] = M[e r + k - 2^i x] + M[s r + k]$;



- Each thread block needs ℓ intermediate results from two rows of M, and ℓ intermediate results to write back

Arbitrary x

- We have work, span and overhead as

$$\begin{aligned} W_{\mathrm{x}} &= (2\,m - \tfrac{1}{2})\,(n + x - 1) \\ S_{\mathrm{x}} &= 2\,x - 1 + \log_2 \tfrac{m}{x} \\ O_{\mathrm{x}} &= \tfrac{3\,(n+x-1)\,(2\,m-x)\,U}{x\,\ell} \end{aligned}$$

- To apply Theorem 1, we have

$$\begin{aligned} N_{\mathrm{x}} &= \tfrac{(n+x-1)\,(2\,m-x)}{x\,\ell} \\ L_{\mathrm{x}} &= \log_2 \tfrac{m}{x} + 1 \\ C_{\mathrm{x}} &= 2\,x - 1 + 3\,U \end{aligned}$$

Comparison of polynomial multiplication algorithms

We replace x by 1 to obtain a "naive" algorithm. Then, we obtain the work ratio and the overhead ratio as

$$\frac{W_1}{W_x} = \frac{n}{n+x-1} \quad \text{and} \quad \frac{O_1}{O_x} = \frac{n\,(2\,m-1)\,x}{(n+x-1)\,(2\,m-x)}$$

Applying Theorem 1 and replacing m by n, when n escapes to infinity, the ratio R is equivalent to

$$R = \frac{(N_1/p + L_1)\cdot C_1}{(N_x/p + L_x)\cdot C_x} \simeq \frac{(1+3\,U)\,x}{2\,x-1+3\,U}$$

- One can assume $3\,U > 1$, which implies that the above ratio is always greater than 1 as soon as $x > 1$ holds
- The algorithm with arbitrary x outperforms the naive one

Experimental results

degree	GPU Plain multiplication	GPU FFT-based multiplication
2^{10}	0.00049	0.0044136
2^{11}	0.0009	0.004642912
2^{12}	0.0032	0.00543696
2^{13}	0.01	0.00543696
2^{14}	0.045	0.00709072

Table: Comparison between plain and FFT-based polynomial multiplications (Moreno Maza and Pan 2010) for balanced pairs ($n = m$) on CUDA.

Multipoint polynomial evaluation

Given a uni-variate polynomial $P = \sum_{j=0}^{n-1} p_j x^j \in K[x]$, with coefficients in the field K, with $n = 2^k$, and evaluation points $u_0, \ldots, u_{n-1} \in K$, compute $P(u_i) = \sum_{j=0}^{n-1} p_j u_i^j$, for $i = 0, \ldots, n-1$.

Polynomial interpolation

Given distinct points $u_0, u_1, \ldots, u_{n-1}$ in a field K and arbitrary values $v_0, v_1, \ldots, v_{n-1} \in K$, compute the unique polynomial $P \in K[x]$ of degree less than $n = 2^k$ that takes the value v_i at the point u_i for all i

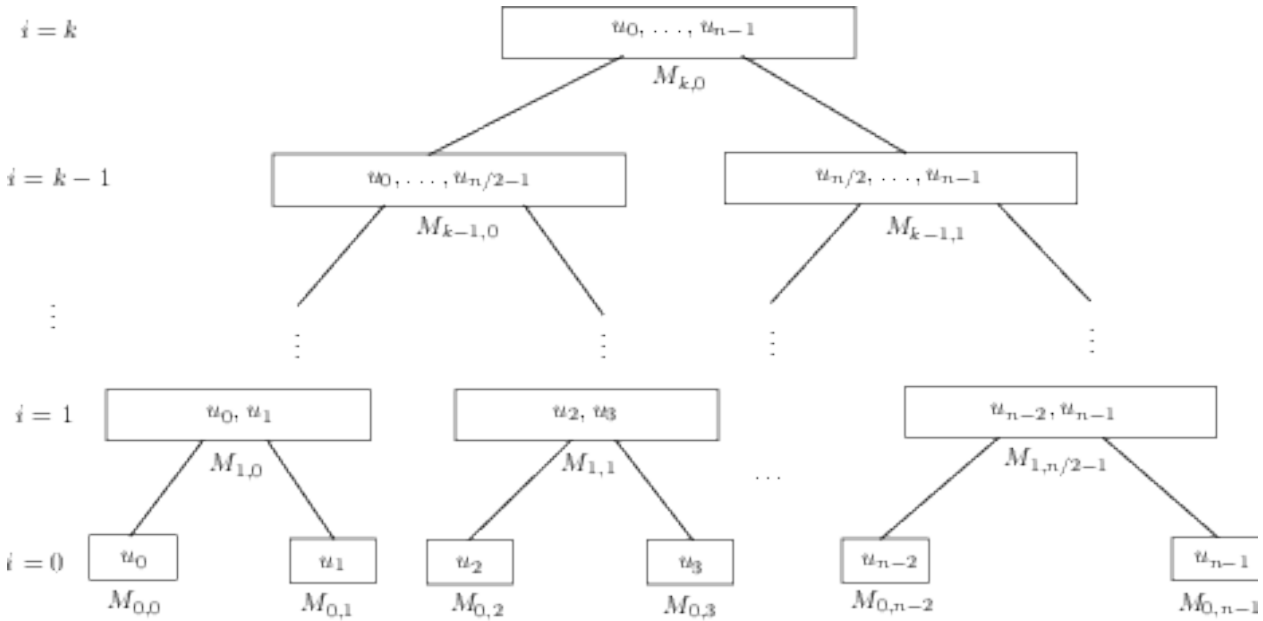
Subproduct tree

Definition

Split the point set $U = \{u_0, \ldots, u_{n-1}\}$ into two halves of equal cardinality and to proceed recursively with each of the two halves. This leads to a binary tree of depth k having the points $u_0, \ldots, u_{n-1}$ as leaves.
Let $m_i = x - u_i$ as above, and define

$$M_{i,j} = m_{j \cdot 2^i} \cdot m_{j \cdot 2^i + 1} \ldots m_{j \cdot 2^i + (2^i - 1)} = \prod_{0 \leq l < 2^i} m_{j \cdot 2^i + l}$$

Subproduct tree



Subproduct tree: construction

Adaptive Algorithm

Let H be a fixed integer with $1 \leq H \leq k$. We call *adaptive algorithm for computing M_n with threshold H* the following procedure:

1. For each level $1 \leq h \leq H$, we compute the subproducts using plain multiplication.
2. Then, for each level $H+1 \leq h \leq k$, we compute the subproducts using FFT-based multiplication.

Implementation: evaluation

Polynomial evaluating

1. $r_0 = P \ rem \ M_{k-1,0}$ and $r_1 = P \ rem \ M_{k-1,1}$
2. Recursively compute $r_0(u_0), \ldots, r_0(u_{n/2-1}), r_1(u_{n/2}), \ldots, r_1(u_{n-1})$

Adaptive top down traversing

Do the remaindering of the polynomials over subproducts, we fix a threshold H:

1. $1 \leq h \leq H$: use plain arithmetic
2. $H+1 \leq h \leq k$: use subinverse tree

We are using reverse and inverse for remaindering

Implementation: subinverse tree

What is subinverse tree?

For the subproduct tree M_n the corresponding *subinverse tree* InvM is a complete binary tree with the same height as M_n and such that, at level i of InvM contains an univariate polynomial $\mathrm{InvM}_{i,j}$ of degree $2^i - 1$ such that for all $0 \leq j < 2^{k-i}$. we have

$$\mathrm{InvM}_{i,j}\,\mathrm{rev}_{2^i+1}(M_{i,j}) \equiv 1 \mod x^{2^i}.$$

Remarks

1. It is used to speedup multi-point evaluation in the degrees where fast division (based on *Newton iteration*) applies.
2. However, subinverse tree is not used lower degrees.

Remarks

1. Algebraic complexity reduced to 50% because of this data structure.

Lagrange interpolation

1. we have $((u_0, v_0), \ldots, (u_{n-1}, v_{n-1}))$
2. $m = \prod_{0 \le i < n}(x - u_i)$, $s_i = \prod_{i \neq j} 1/(u_i - u_j)$
3. $f = \sum_{i=0}^{n} v_i s_i m/(x - u_i)$

Note: $1/s_i = m'(u_i)$, $P = M_{k-1,0} P_0 + M_{k-1,1} P_1$

Adaptive Algorithm

For computing intermediate results, we fix a threshold H:

1. $1 \le h \le H$: use plain multiplication
2. $H + 1 \le h \le k$: use FFT-based multiplication

Experimentation

	Evaluation			Interpolation		
Deg.	GPU	FLINT	SpeedUp	GPU	FLINT	SpeedUp
8	0.0310	0	0	0.0328	0	0
9	0.0623	0	0	0.0669	0	0
10	0.0843	0	0	0.0968	0.01	0.1032
11	0.1012	0.01	0.0987	0.1202	0.01	0.0831
12	0.1361	0.02	0.1468	0.1671	0.03	0.1794
13	0.1580	0.07	0.4429	0.1963	0.09	0.4584
14	0.2034	0.17	0.8354	0.2548	0.22	0.8631
15	0.2415	0.41	1.6971	0.3073	0.53	1.7242
16	0.3126	0.99	3.1666	0.4026	1.26	3.1294
17	0.4285	2.33	5.4375	0.5677	2.94	5.1780
18	0.7106	5.43	7.6404	0.9034	6.81	7.5379
19	1.0936	12.63	11.5484	1.3931	15.85	11.3768
20	1.9412	29.2	15.0420	2.4363	36.61	15.0268
21	3.6927	67.18	18.1923	4.5965	83.98	18.2702
22	7.4855	153.07	20.4486	9.2940	191.32	20.5851
23	15.796	346.44	21.9321	19.6923	432.13	21.9441

Overview

1. We integrated CUDA support for computing polynomial gcd, division and multiplication into a bivariate solver over finite fields
2. This bivariate solver is written in C and is part of the `cumodp` library.
3. In particular, it relies on CUDA code for computing subresultant chains developed by Dr. Wei Pa nd Marc Moreno Maza.
4. All these features together make bivariate solver very powerful.

system	Pure C	C with CUDA support	speed-up
dense-70	5.22	0.50	10.26
dense-80	6.63	0.77	8.59
dense-90	8.39	1.16	7.19
dense-100	19.53	1.80	10.79
sparse-70	0.89	0.31	2.81
sparse-80	3.64	1.18	3.09
sparse-90	3.13	0.92	3.40
sparse-100	8.86	1.20	7.38

Concluding remarks (1/2)

Around sparse matrix vector multiplication

- We proposed new algorithms for improving the data locality of basic routines dealing with vectors and sparse matrices.
- In each case, we re-arrange the input data and amortize the cost of this re-arrangement against the cost of calculations with the input data.
- We provide cache complexity analysis whose favorable results are confirmed experimentally.

Adaptive algorithms for subproduct tree techniques on GPUs

- We propose parallel algorithms for performing subproduct tree construction, evaluation and interpolation and report on their implementation on many-core GPUs
- We enhance the traditional algorithms for polynomial evaluation and interpolation based on subproduct-trees, by introducing the notion of a subinverse tree.
- For subproduct-tree operations, we demonstrate the importance of adaptive algorithms. That is, algorithms that adapt their behavior to the available computing resources.
- In particular, we combine parallel plain arithmetic and parallel fast arithmetic.

Concluding remarks (2/2)

Plain arithmetic on GPUs

- We presented a model of multithreaded computation, combining the fork-join and SIMD parallelisms, with an emphasis on estimating parallelism overheads, so as to reduce scheduling and communication costs in GPU programs.
- We have applied this model and successfully reduced parallelsim overheads for seevral basic routines in polynomial algebra.
- For polynomial multiplication, our theoretical analysis allows us to reduce parallelism overheads due not only to data transfer but also to code divergence.
- For the Euclidean algorithm, our running time estimates match those obtained with the Systolic VLSI Array Model (Brent & Kung, 1984). Meanwhile, our CUDA code implementing this optimized Euclidean algorithm runs within the same estimate analyzed by our model for input polynomials with degree up to 100,000.
- Finally, our order of magnitude estimates for the program parameter of radix sort agrees with the empirical results of (Satish, Harris, and Garland, 2009).
- All our GPU code is freely available in source at `www.cumodp.org`

Thank you