A Many-core Machine Model for Designing Algorithms with Minimum Parallelism Overheads

Sardar Anisul Haque Marc Moreno Maza Ning Xie

University of Western Ontario

American University of Beirut
June 16-18, 2014

Plan

Parallelism Overheads and Models of Computations

Why is my parallel program not reaching linear speedup? not scaling?

- The algorithm could lack of parallelism ...
- The architecture could suffer from limitations ...
- The program does not expose enough parallelism ...
- Or the concurrency platform suffers from overheads (such as communication and synchronization costs)!

Challenges for models of computations

- Retaining the features of actual computers that have a dominant impact on program performance is hard
- Using several complexity measures (work, span, cache complexity) is necessary, but
- how to combine those complexity measures in order to select the best algorithm among several candidates for a given problem?
- Parallelism overheads are often ignored or included with other performance counters.

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

A many-core machine model:

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

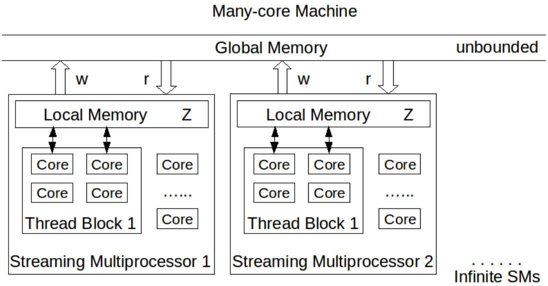
How to use this model?

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 $\mathcal{P}(s_o)$.

Plan

MMM: characteristics

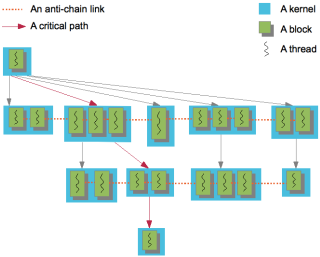


Architecture:

- 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

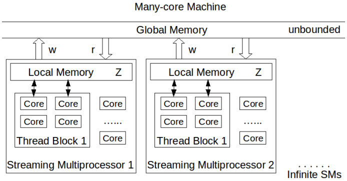
MMM: programs

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: Execution model



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 of the SM running the thread-block
- 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

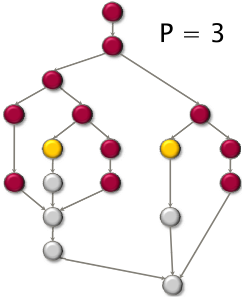
$$O(\mathcal{P}) = \sum_{\alpha} O(\alpha)$$

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

Graham - Brent Theorem: original version



- In any *greedy schedule*, there are two types of steps:
  - **complete step**: There are at least p strands that are ready to run. The greedy scheduler selects any p of them and runs them.
  - **incomplete step**: There are strictly less than p threads that are ready to run. The greedy scheduler runs them all.
- *For any greedy schedule, we have* $T_p \leq T_1/p + T_\infty$

MMM: complexity measures

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

Corollary

Let K *be the maximum number of thread blocks along an anti-chain of the thread-block DAG of $\mathcal{P}$. Then the running time $T_{\mathcal{P}}$ of the program $\mathcal{P}$ satisfies:*

$$T_{\mathcal{P}} \leq (N(\mathcal{P})/\mathrm{K} + L(\mathcal{P}))C(\mathcal{P}) \tag{1}$$

This estimate does not depend on the number of SMs in use to execute $\mathcal{P}$.

Plan

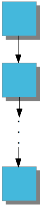
Plain division for polynomials

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. We shall consider two approaches:

- a naive division algorithm
- an division algorithm optimized in terms of parallelism overheads.

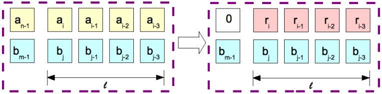
We assume that

- $n \geq m$

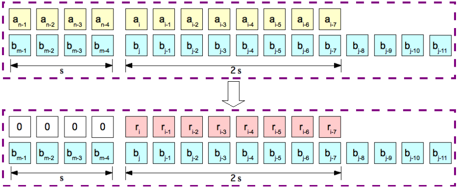


Plain division algorithms

Naive Division Algorithm



Optimized Division Algorithm



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

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

Analysis for the naive division algorithm

- In each kernel call, each thread computes one coefficient of an intermediate remainder polynomial by means of one multiplication and one subtraction in the coefficient field $\mathbb{K}$.
- Let ℓ be the number of threads in a thread-block, we note that each kernel uses $\lceil \frac{m}{\ell} \rceil$ thread-blocks.
- We observe that each thread of a kernel reads/writes 3 to 5 words

Since each thread-block performs $2\,\ell + 1$ arithmetic operations and each thread makes at most 5 accesses to the global memory, we have

$$W_{\mathrm{nai}} = \frac{(n - m + 1)\, m\, (2\,\ell + 1)}{\ell},\ S_{\mathrm{nai}} = 3\,(n - m + 1),$$

and

$$O_{\mathrm{nai}} = \frac{5\,(n - m + 1)\, m\, U}{\ell}.$$

Moreover, the quantities $N(\mathcal{P})$, $L(\mathcal{P})$ and $C(\mathcal{P})$ are respectively given by

$$N_{\mathrm{nai}} = \frac{(n - m + 1)\, m}{\ell},\ L_{\mathrm{nai}} = (n - m + 1) \ \text{ and } \ C_{\mathrm{nai}} = 3 + 5\,U.$$

Analysis for the optimized division algorithm

- Fixing $s \geq 1$, each kernel call performs at most s division steps
- To this end, each thread-block
  - uses $3\,s$ threads,
  - loads the coefficients of X^d, X^{d-1}, …, X^{d-s+1} from a (resp. b), that we call the s-*head* of a (resp. b), where d is the degree of a (resp. b),
  - loads $2\,s$ (resp. $3\,s$) consecutive coeff. of a (resp. b), say X^{d_1}, X^{d_1-1}, …, X^{d_1-2s+1} (X^{d_2}, X^{d_2-1}, …, X^{d_2-3s+1}) for some integer $d_1 > 0$ (resp. $d_2 > 0$) which depends on the thread and thread-block IDs.

Since each thread makes at most 9 accesses to the global memory, we have the following estimates for the work, span and overhead of the optimized algorithm

$$W_{\rm opt} = \frac{(n-m+1)\,m\,(9\,s+1)}{4\,s},\ S_{\rm opt} = 3\,(n-m+1),$$

and

$$O_{\rm opt} = \frac{9(n-m+1)m\,U}{2\,s^2}.$$

Comparison of the two plain division algorithms (1/2)

Inequality constraints

We replace ℓ and s by $Z/2$ and $Z/7$, respectively, since 2ℓ or $7s$ coefficients must fit into the local memory, that is, $2\ell \leq Z$ and $7s \leq Z$.

We obtain the work ratio and the overhead ratio as

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

Applying the corollary of Theorem 1,

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

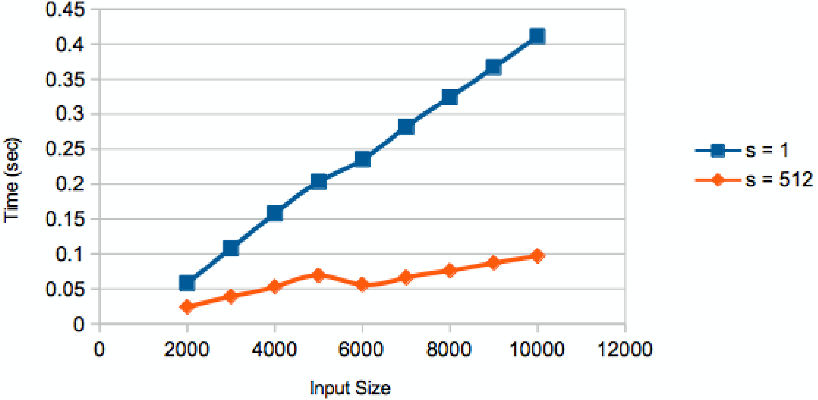
Comparison of the two plain division algorithms (1/2)

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 for the division and the Euclidean algorithm



Experimental results for the division and the Euclidean algorithm

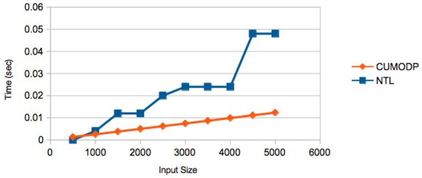


Figure: CUMODP plain polynomial division vs NTL FFT-based (asymptotically fast) polynomial division.

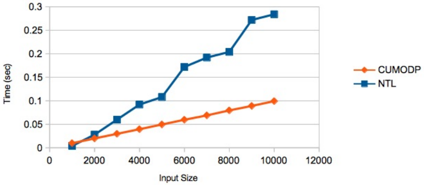


Figure: CUMODP plain Euclidean algorithm vs NTL FFT-based polynomial GCD.

Plan

Plain multiplication for polynomials (1/2)

Notations

- Let $\mathbb{K}$ be a field and $a, b \in \mathbb{K}[X]$ be two univariate polynomials over $\mathbb{K}$ and with variable X.
- Let n and m be positive integers such that $\deg(a) = n - 1$ and $\deg(b) = m - 1$.
- Our multiplication algorithm is based on the well-known *long multiplicatio*; we consider two approaches:
  - a naive division algorithm
  - an division algorithm optimized in terms of parallelism overheads.

Principles

- During the *multiplication phase*, every coefficient of a is multiplied with every coefficient of b;
- the resulting products are accumulated in an intermediate array, denoted by M.
- Then, during the *addition phase*, these accumulated products are added together to form the polynomial f.
-

Plain multiplication for polynomials (2/2)

Principles (recall)

- During the *multiplication phase*, every coefficient of a is multiplied with every coefficient of b;
- the resulting products are accumulated in an intermediate array, denoted by M.
- Then, during the *addition phase*, these accumulated products are added together to form the polynomial f.
-

Program parameter

For this application, the program parameter s an integer $s > 0$, representing for each thread-block:

- the number of coefficients of b to be multiplied by a number of coefficients of a in the coefficients multiplication phase,
- as well as the number of sums per thread in the addition phase.

As before, we denote by ℓ the number of threads per thread-block.

Plain multiplication algorithm (1/3)

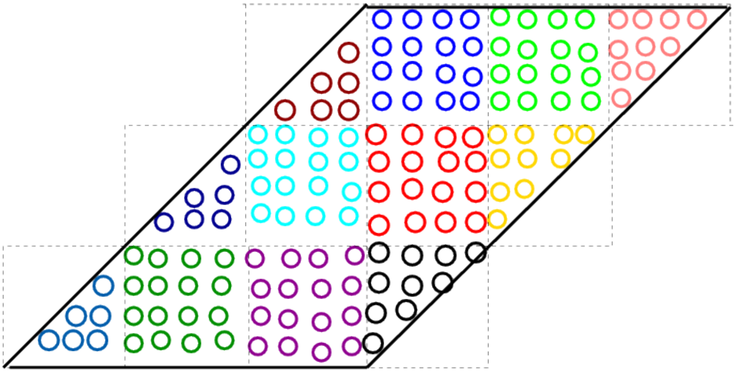
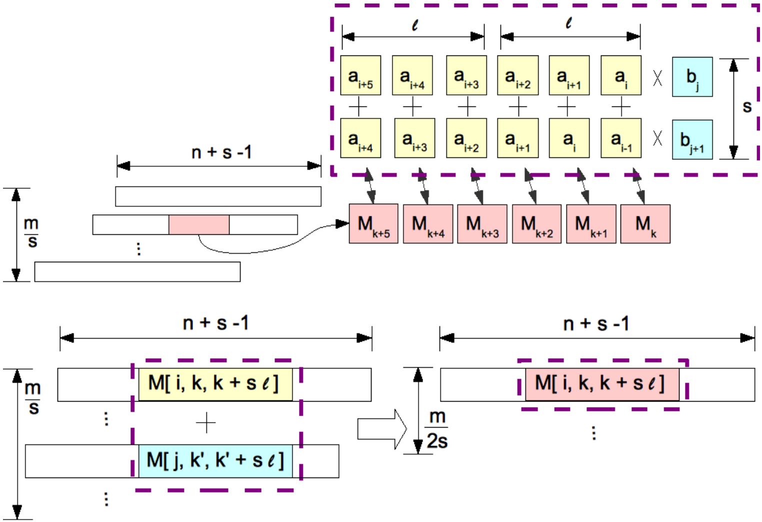
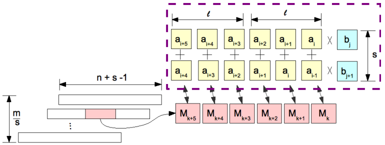


Figure: Each rectangular is computed by one thread block.

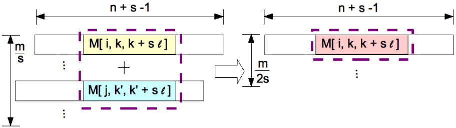
Plain multiplication algorithm (2/3)



Multipliaction phase



Addition phase



Each thread-block

- reads $s\,\ell + s - 1$ coefficients of a and s coefficients of b
- computes $\ell\,s^2$ products, followed by $\ell\,s\,(s-1)$ of additions

Thus, each thread-block contributes $s\,\ell$ *partial sums* to the 2-D array M, whose format is $x \cdot y$, where $x = \frac{m}{s}$ and $y = n + s - 1$.

- The x rows of the auxiliary array M are added pairwise in $\log_2 x$ parallel steps.
- When adding rows i and j (for $i < j$) at a given parallel step, each thread-block loads $s\,\ell$ elements of $M[i]$ and $M[j]$, respectively, and then adds $M[j]$ to $M[i]$.

Analysis for the plain multipliaction algorithm (1/4)

Observations

- We denote by $W_{\rm s}$, $S_{\rm s}$ and, $O_{\rm s}$, the work, span and overhead, respectively for the program with parameter s.
- Considering any thread-block of the multiplication phase, we notice that s coefficients of b and $s\,\ell + s - 1$ coefficients of a are loaded and $s\,\ell$ results are written back to global memory.
- Hence $2\,s\,\ell + 2\,s - 1$ coefficients must fit into local memory, that is, we have $2\,s\,\ell + 2\,s - 1 \leq Z$.

We obtain the following estimates:

$$W_s = (2\,m - \frac{1}{2})\,(n + s - 1),\ \ S_s = 2\,s^2 + s\,\log_2 \frac{m}{s} - s \tag{2}$$

and

$$O_s = \frac{(n + s - 1)\,(5\,m\,s + 2\,m - 3\,s^2)\,U}{s^2\,\ell} \tag{3}$$

Analysis for the plain multipliaction algorithm (2/4)

Graham-Brent Theorem Coefficients

We obtain the quantities characterizing the thread block DAG that are required in order to apply Graham-Brent Theorem:
$N_s = \frac{(n+s-1)\,(2\,m-s)}{s^2\,\ell}$, $L_s = \log_2 \frac{m}{s} + 1$ and $C_s = s\,(2\,s-1) + 2\,U\,(s+1)$.

Ratios

- We set $s = 1$, and view the resulting algorithm as a "naive one".
- The work ratio $W_1/W_s = \frac{n}{n+s-1}$, is asymptotically constant as n escapes to infinity.
- The span ratio $S_1/S_s = \frac{\log_2 m+1}{s\,(\log_2(m/s)+2\,s-1)}$ shows that S_s grows asymptotically with s.
- The parallelism overhead ratio, letting $m = n$:

$$\frac{O_1}{O_s} = \frac{n\,s^2\,(7\,n-3)}{(n+s-1)\,(5\,n\,s+2\,n-3\,s^2)}. \tag{4}$$

  We observe that, as n escape to infinity, this latter ratio is asymptotically equivalent to s.

Analysis for the plain multipliaction algorithm (3/4)

Determining s

- Applying the corollary, let R be the ratio of the running time estimate between the naive algorithm and that for an arbitrary s. We obtain

$$R = \frac{(n \log_2 n + 3n - 1)(1 + 4U)}{(n \log_2 \frac{n}{s} + 3n - s)(2Us + 2U + 2s^2 - s)}, \tag{5}$$

  which is essentially

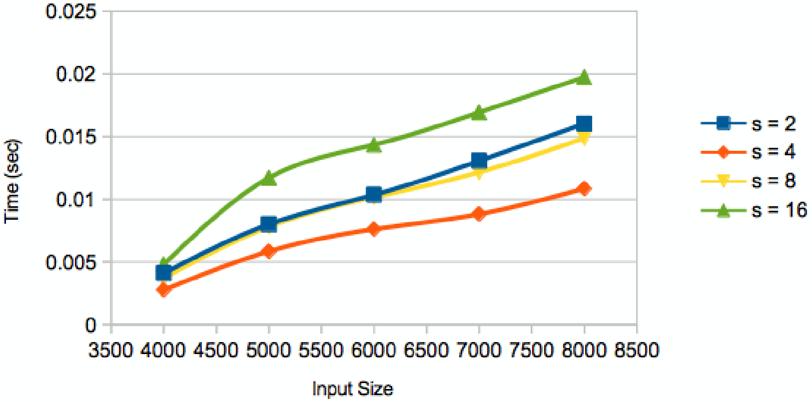
$$\frac{2 \log_2 n}{s \log_2 (n/s)}.$$

- This latter ratio is smaller than 1, such that the "initial" algorithm (that is for $s = 1$) performs better.
- This also indicates that increasing s makes the algorithm performance worse.

Analysis for the plain multipliaction algorithm (4/4)

Experimentally

In practice, as shown in tables, setting $s = 4$ performs best, while with larger s, the running time becomes slower, which is coherent with our theoretical analysis.



Experimental results for the plain multipliaction (1/2)

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 for balanced pairs ($n = m$) on CUDA.

Experimental results for the plain multipliaction (1/2)

degree(A)	degree(B)	GPU Plain multiplication
2^{10}	2^8	0.00041
2^{11}	2^8	0.0005
2^{11}	2^{10}	0.00073
2^{12}	2^8	0.00057
2^{12}	2^{10}	0.0011
2^{13}	2^8	0.00074
2^{13}	2^{10}	0.0018
2^{13}	2^{12}	0.0061
2^{14}	2^8	0.0010
2^{14}	2^{10}	0.0031
2^{14}	2^{12}	0.011
2^{14}	2^{13}	0.02

Table: Computation time for plain multiplication on CUDA for unbalance pairs ($n \neq m$).

Plan

Summary

www.cumodp.org