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

Sardar Anisul Haque, Marc Moreno Maza and Ning Xie
University of Western Ontario, London, Ontario, Canada

February 28, 2015

Abstract

We present a model of multithreaded computation with an emphasis on estimating parallelism overheads of programs written for modern many-core architectures. We establish a Graham-Brent theorem for this model so as to estimate execution time of programs running on a given number of streaming multiprocessors. We evaluate the benefits of our model with fundamental algorithms from scientific computing. For two case studies, our model is used to minimize parallelism overheads by determining an appropriate value range for a given program parameter. For the others, our model is used to compare different algorithms solving the same problem. In each case, the studied algorithms were implemented and the results of their experimental comparison are coherent with the theoretical analysis based on our model.

1 Introduction

Designing efficient algorithms targeting hardware accelerators (multi-core processors, graphics processing units (GPUs), field-programmable gate arrays) creates major challenges for computer scientists. A first difficulty is to define models of computation retaining the computer hardware characteristics that have a dominant impact on program performance. That is, in addition to specify the appropriate complexity measures, those models must consider the relevant parameters characterizing the abstract machine executing the algorithms to be analyzed, A second difficulty is, for a given model of computation, to combine its complexity measures so as to determine the "best" algorithm among different possible algorithmic solutions to a given problem.

In the fork-join concurrency model [2] two complexity measures, the work T_1 and the span T_∞, and one machine parameter, the number P of processors, are combined into a running time estimate, like in the Graham-Brent theorem [2, 6] or the results of Blumofe & Leiserson [3]. The Graham-Brent theorem states that the running time T_P on P processors satisfies $T_\mathsf{P} \leq T_1/\mathsf{P} + T_\infty$. A refinement of this theorem supports the implementation (on multi-core architectures) of the parallel performance analyzer `Cilkview` [10]. In this context, the running time T_P is bounded in expectation by $T_1/\mathsf{P} + 2\delta\widehat{T_\infty}$, where δ is a constant (called the *span coefficient*) and $\widehat{T_\infty}$ is the burdened span, which captures parallelism overheads due to scheduling and synchronization.

The PRAM (parallel random-access machine) model [21, 5] has also been enhanced [1] so as to integrate communication delay into the computation time. However, a PRAM abstract machine consists of an unbounded collection of RAM processors, whereas a many-core GPU holds a collection of streaming multiprocessors (SMs). Hence, applying the PRAM model to GPU programs may fail to fully capture the impact of data transfer between the SMs and the global memory of the device.

Ma, Agrawal and Chamberlain [13] introduce the TMM (Threaded Many-core Memory) model which retains many important characteristics of GPU-type architectures as machine parameters, such as memory access width and hardware limit on number of threads per core. In TMM analysis, the running time of an algorithm is estimated by choosing the maximum quantity among the work, span and amount of memory accesses. Such running time estimates depend on the machine parameters.

Many works, such as [14, 12], targeting code optimization and performance prediction of GPU programs are related to our work. However, these papers do not define an abstract model in support of the analysis of algorithms.

In this paper, we propose a many-core machine (MCM) model which aims at (1) tuning program parameters to minimize parallelism overheads of algorithms targeting GPU-like architectures as well as (2) comparing different algorithms independently of the targeted hardware device. In the design of this model, we insist on the following features:

- *Two-level DAG programs.* Defined in Section 2, this aspect captures the two levels of parallelism (fork-join and SIMD) of heterogeneous programs (like a CilkPlus program using `#pragma simd` [18] or a CUDA program with the so-called dynamic parallelism [17]).

- *Parallelism overhead.* We introduce this complexity measure in Section 2.3 with the objective of analyzing communication and synchronization costs.

- *A Graham-Brent theorem.* We combine three complexity measures (work, span and parallelism overhead) and one machine parameter (data transfer throughput) in order to estimate the running time, based on Theorem 1 in Section 2.4, of an MCM program on P streaming multiprocessors. However, as we shall see through a series of case study, this machine parameter has no influence on the comparison of algorithms.

Our model extends both the fork-join concurrency model and PRAM models, with an emphasis on parallelism overheads resulting from communication and synchronization.

We sketch below how, in practice, we use this model to tune a program parameter so as to minimize parallelism overheads of programs targeting many-core GPUs. Consider an MCM program $\mathcal{P}$, that is, an algorithm expressed in the MCM model. Assume that a program parameter s (like the amount of data transfered by one thread) can be arbitrarily chosen within some range $\mathcal{S}$ while preserving the specifications of $\mathcal{P}$. Let s_0 be a particular value of s which corresponds to an instance $\mathcal{P}_0$ of $\mathcal{P}$, which, in practice, is seen as an initial version of the algorithm to be optimized.

We consider the ratios of the work, span, and parallelism overhead given by $\mathrm{W}_{\mathcal{P}}(s_0)/\mathrm{W}_{\mathcal{P}}(s)$, $\mathrm{S}_{\mathcal{P}}(s_0)/\mathrm{S}_{\mathcal{P}}(s)$ and $\mathrm{O}_{\mathcal{P}}(s_0)/\mathrm{O}_{\mathcal{P}}(s)$. Assume that, when s varies within $\mathcal{S}$, the work ratio and span ratio stay within $O(s)$ ($\Theta(1)$ is often the case), but the ratio of the parallelism overhead reduces by a factor in $\Theta(s)$. Thereby, we determine a value $s_{\min} \in \mathcal{S}$ maximizing the parallelism overhead ratio. Next, we use our version of Graham-Brent theorem (more

precisely, we use Corollary 1) to check whether the upper bound for the running time of $\mathcal{P}(s_{\min})$ is less than that of $\mathcal{P}(s_o)$. If this holds, we view $\mathcal{P}(s_{\min})$ as a solution of our problem of algorithm optimization (in terms of parallelism overheads).

To demonstrate and evaluate the benefits of our model, we apply it successfully to five fundamental algorithms [1] in scientific computing, see Sections 3 to 5. These five algorithms are the Euclidean algorithm, Cooley & Tukey and Stockham fast Fourier transform algorithms, and the plain and FFT-based univariate polynomial multiplication algorithms. Other applications of our model can be found in the PhD thesis [8] of the first Author as well as in [9].

Following the strategy described above for algorithm optimization, our model is used to tune a program parameter in the case of the Euclidean algorithm and the plain multiplication algorithm. Next, our model is used to compare two different fast Fourier transform algorithms and then two different univariate polynomial multiplication algorithms. In each case, work, span and parallelism overhead are evaluated so as to obtain a running time estimates via our Graham-Brent theorem and select a proper algorithm.

2 A many-core machine model

The model of parallel computations presented in this paper aims at capturing communication and synchronization overheads of programs written for modern many-core architectures. One of our objectives is to optimize algorithms by techniques like reducing redundant memory accesses. The reason for this optimization is the fact that, on actual GPUs, global memory latency is approximately 400 to 800 clock cycles. This memory latency, when not properly taken into account, may have a dramatically negative impact on program performance. Another objective of our model is to compare different algorithms targeting implementation on GPUs without taking hardware parameters into account.

As specified in Sections 2.1 and 2.2, our many-core machine (MCM) model retains many of the characteristics of modern GPU architectures and programming models, like CUDA or OpenCL. However, in order to support algorithm analysis with an emphasis on parallelism overheads, as defined in Section 2.3 and 2.4, the MCM abstract machines admit a few simplifications and limitations with respect to actual many-core devices.

2.1 Characteristics of the abstract many-core machines

Architecture. An MCM abstract machine possesses an unbounded number of *streaming multiprocessors* (SMs) which are all identical. Each SM has a finite number of processing cores and a fixed-size private memory. An MCM machine has a two-level memory hierarchy, comprising an unbounded global memory with high latency and low throughput while private memories have low latency and high throughput.

Programs. An MCM *program* is a directed acyclic graph (DAG) whose vertices are kernels (defined hereafter) and edges indicate serial dependencies, similarly to the instruction stream DAGs of the fork-join multithreaded concurrency model. A *kernel* is an SIMD (single instruction, multiple data) program capable of branches and decomposed into a

[1] Our algorithms are implemented in CUDA and publicly available with benchmarking scripts from `http://www.cumodp.org/`.

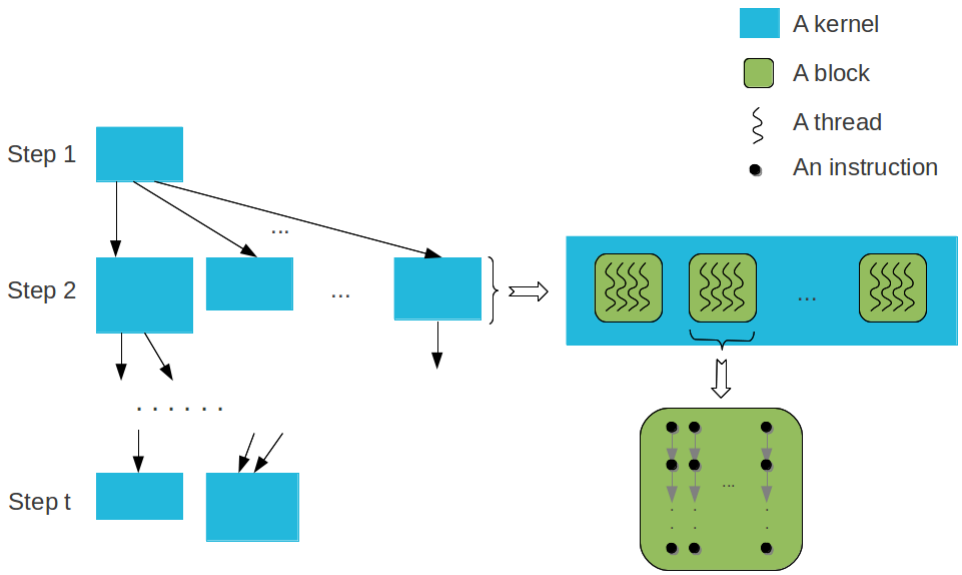

Figure 1: Overview of a many-core machine program

number of thread-blocks. Each *thread-block* is executed by a single SM and each SM executes a single thread-block at a time. Similarly to a CUDA program, an MCM program specifies for each kernel the number of thread-blocks and the number of threads per thread-block, following the same extended function-call syntax. Figure 1 depicts the different types of components of an MCM program.

Scheduling and synchronization. At run time, an MCM machine schedules thread-blocks (from the same or different kernels) onto SMs, based on the dependencies specified by the edges of the DAG and the hardware resources required by each thread-block. Threads within a thread-block can cooperate with each other via the private memory of the SM running the thread-block. Meanwhile, thread-blocks interact with each other via the global memory. In addition, threads within a thread-block are executed physically in parallel by an SM. Moreover, the programmer cannot make any assumptions on the order in which thread-blocks of a given kernel are mapped to the SMs. Hence, MCM programs run correctly on any fixed number of SMs.

Memory access policy. All threads of a given thread-block can access simultaneously any memory cell of the private memory or the global memory: read/write conflicts are handled by the CREW (concurrent read, exclusive write) policy. However, read/write requests to the global memory by two different thread-blocks cannot be executed simultaneously. In case of simultaneous requests, one thread-block is chosen randomly and served first, then the other is served.

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

U: Time (expressed in clock cycles) to transfer one machine word between the global memory and the private memory of any SM, that is, $U > 0$.

Z: Size (expressed in machine words) of the private memory of any SM, which sets up an upper bound on several program parameters.

The private memory size Z unifies different characteristics of an SM and, thus, of a thread-block. Indeed, each of the following quantities is at most equal to Z: the number of threads

of a thread-block or the number of words in a data transfer between the global memory and the private memory of a thread-block.

On the other hand, the quantity $1/U$ is a throughput measure and has the following property. If α and β are the numbers of words respectively read and written to the global memory by one thread of a thread-block B and ℓ be the number of threads per thread-block, then the total time T_D spent in data transfer between the global memory and the private memory of an SM executing B satisfies

$$\begin{array}{rcl} T_D & \leq & (\alpha + \beta)\, U, \text{ if coalesced accesses occur;} \\ & \text{or} & \ell\,(\alpha + \beta)\, U, \text{ otherwise.} \end{array} \tag{1}$$

We observe that, on actual machines, some hardware characteristics may reduce data transfer time, for instance, coalesced accesses to global memory and fast context switching to hide data transfer latency. Other hardware characteristics, i.e. partition camping, may increase data transfer time. As an abstract machine, the MCM aims at capturing either the best or the worst scenario for data transfer time of a thread-block, which are given by Relation (1).

Relation (1) calls for another comment. One could expect the introduction of a third machine parameter, say V, which would be the time to execute one *local operation* (arithmetic operation, read/write in the private memory), such that, if σ is the total number of local operations performed by one thread of a thread-block B, then the total time T_A spent in local operations by an SM executing B would satisfy $T_A \leq \sigma V$. Therefore, for the total running time T of the thread-block B, we would have

$$T \;=\; T_A + T_D \;\leq\; \sigma\, V + \epsilon\,(\alpha + \beta)\, U,$$

where ϵ is either 1 or ℓ. Instead of introducing this third machine parameter V, we let $V = 1$. In other words, U can be understood as the ratio between the time to transfer a machine word and the time to execute a local operation.

2.2 Many-core machine programs

Recall that each MCM program $\mathcal{P}$ is a DAG $(\mathcal{K}, \mathcal{E})$, called the *kernel DAG* of $\mathcal{P}$, where each node $K \in \mathcal{K}$ represents a kernel, and each edge $E \in \mathcal{E}$ records the fact that a kernel call must precede another kernel call. In other words, a kernel call can be executed provided that all its predecessors in the DAG $(\mathcal{K}, \mathcal{E})$ have completed their execution.

Synchronization costs. Recall that each kernel decomposes into thread-blocks and that all threads within a given kernel execute the same serial program, but with possibly different input data. In addition, all threads within a thread-block are executed physically in parallel by an SM. It follows that MCM kernel code needs no synchronization statement, like CUDA's `__syncthreads()`. Consequently, the only form of synchronization taking place among the threads executing a given thread-block is that implied by code divergence [7]. This latter phenomenon can be seen as parallelism overhead. Further, an MCM machine handles code divergence by eliminating the corresponding conditional branches via code replication [19], and the corresponding cost will be captured by the complexity measures (work, span and parallelism overhead) of the MCM model.

Scheduling costs. Since an MCM abstract machine has infinitely many SMs and since the kernel DAG defining an MCM program $\mathcal{P}$ is assumed to be known when $\mathcal{P}$ starts to

execute, scheduling $\mathcal{P}$'s kernels onto the SMs can be done in time $O(\Gamma)$ where Γ is the total length of $\mathcal{P}$'s kernel code. Thus, we shall neglect those costs in comparison to the costs of transferring data between SMs' private memories and the global memory. We also note that assuming that, the kernel DAG is known when $\mathcal{P}$ starts to execute, allows us to focus on parallelism overheads resulting from this data transfer. Extending MCM machines to support programs whose instruction stream DAGs unfold dynamically at run time as in [10] is left for future work.

Thread-block DAG. Since each kernel of the program $\mathcal{P}$ decomposes into finitely many 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 $\mathcal{P}$ and such that (B_1, B_2) is an edge if B_1 is a thread-block of a kernel preceding the kernel of the thread-block B_2 in $\mathcal{P}$. This second graph defines two important quantities:

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

$\mathrm{L}(\mathcal{P})$**:** critical path length (where length of a path is the number of edges in that path) in the thread-block DAG of $\mathcal{P}$.

2.3 Complexity measures for the many-core machine model

Consider an MCM program $\mathcal{P}$ given by its kernel DAG $(\mathcal{K}, \mathcal{E})$. Let $K \in \mathcal{K}$ be any kernel of $\mathcal{P}$ and B be any thread-block of K. We define the *work* of B, denoted by $\mathrm{W}(B)$, as the total number of local operations performed by all threads of B. We define the *span* of B, denoted by $\mathrm{S}(B)$, as the maximum number of local operations performed by a thread of B. Let α and β be the maximum numbers of words read and written (from the global memory) by a thread of B, and ℓ be the number of threads per thread-block. Then, we define the *overhead* of B, denoted by $\mathrm{O}(B)$, as

$$(\alpha + \beta)\, U, \text{ if memory accesses can be coalesced;}$$

$$\text{or } \ell\,(\alpha + \beta)\, U, \text{ otherwise.}$$

Next, the *work* (resp. *overhead*) $\mathrm{W}(K)$ (resp. $\mathrm{O}(K)$) of the kernel K is the sum of the works (resp. overheads) of its thread-blocks, while the *span* $\mathrm{S}(K)$ of the kernel K is the maximum of the spans of its thread-blocks.

We consider now the entire program $\mathcal{P}$. The *work* $\mathrm{W}(\mathcal{P})$ of $\mathcal{P}$ is defined as the total work of all its kernels

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

Regarding the graph (K, E) as a weighted-vertex graph where the weight of a vertex $K \in \mathcal{K}$ is its span $\mathrm{S}(K)$, we define the weight $\mathrm{S}(\gamma)$ of any path γ from the first executing kernel to the last executing kernel as $\mathrm{S}(\gamma) = \sum_{K \in \gamma} \mathrm{S}(K)$. Then, we define the *span* $\mathrm{S}(\mathcal{P})$ of the program $\mathcal{P}$ as

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

Finally, we define the *overhead* $\mathrm{O}(\mathcal{P})$ of the program $\mathcal{P}$ as the total overhead of all its kernels

$$\mathrm{O}(\mathcal{P}) = \sum_{K \in \mathcal{K}} \mathrm{O}(K).$$

Observe that, according to Mirsky's theorem [15], the number π of parallel steps in $\mathcal{P}$ (i.e. anti-chains in $(\mathcal{K}, \mathcal{E})$) is equal to the maximum length of a path in $(\mathcal{K}, \mathcal{E})$ from the first executing kernel to the last executing kernel.

2.4 A Graham-Brent theorem with parallelism overhead

Theorem 1 *We have the following estimate for the running time T_P of the program $\mathcal{P}$ when executed on P SMs:*

$$T_{\mathsf{P}} \leq (\mathrm{N}(\mathcal{P})/\mathsf{P} + \mathrm{L}(\mathcal{P}))\mathrm{C}(\mathcal{P}) \tag{2}$$

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

The proof is similar to that of the original result. One observes that the total number of *complete steps* (for which P thread-blocks can be scheduled by a greedy scheduler) is at most $\mathrm{N}(\mathcal{P})/\mathsf{P}$ while the number of *incomplete steps* is at most $\mathrm{L}(\mathcal{P})$. Finally, $\mathrm{C}(\mathcal{P})$ is an obvious upper bound for the running time of every step, complete or incomplete.

The proof of the following corollary follows from Theorem 1 and from the fact that costs of scheduling thread-blocks onto SMs are neglected.

Corollary 1 *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:*

$$T_{\mathcal{P}} \leq (\mathrm{N}(\mathcal{P})/\mathsf{K} + \mathrm{L}(\mathcal{P}))\mathrm{C}(\mathcal{P}) \tag{3}$$

As we shall see in Sections 3 through 5, Corollary 1 allows us to estimate the running time of an MCM program as a function of the number ℓ of threads per thread-block, the single machine parameter U and the thread-block DAG of $\mathcal{P}$. Thus, the dependence on the machine parameter Z (the size of a private memory) is only through inequalities specifying upper bounds for ℓ. In addition, in each of case studies, there is no need to make any assumptions (like inequality constraints) on the machine parameter U.

3 The Euclidean algorithm

Our first application of the MCM model deals with a multithreaded algorithm for computing the greatest common divisor (GCD) of two univariate polynomials. To specify notations, let $\mathbb{K}$ be a field of coefficients (like the finite field $\mathbb{Z}/p\mathbb{Z}$ of prime characteristic p) and $\mathbb{K}[X]$ be the set of all univariate polynomials with coefficients in $\mathbb{K}$.

Our approach is based on the Euclidean algorithm, that the reader can review in Chapter 4 in [11]. Given a positive integer s, we proceed by repeatedly calling a subroutine (see Algorithm 1) which takes as input a pair (a, b) of polynomials in $\mathbb{K}[X]$, with $\deg(a) \geq \deg(b) > 0$, and returns another pair (a', b') of polynomials in $\mathbb{K}[X]$, such that $\gcd(a, b) = \gcd(a', b')$ and, either $b' = 0$ (in which case we have $\gcd(a, b) = a'$), or we have $\deg(a') + \deg(b') \leq \deg(a) + \deg(b) - s$. We will take advantage of our MCM model to tune the program parameter s in order to obtain an optimized multithreaded version of the Euclidean algorithm.

Let n and m be positive integers such that $\deg(a) = n - 1$ and $\deg(b) = m - 1$, assuming $n \geq m$. We use a dense representation for encoding the polynomials a and b. Thus, n and

m are the number of coefficients (zero or not) of a and b, respectively. Algorithm 1 uses two arrays of size to represent a and b, where the coefficient of the term of a (resp. b) in degree i is stored in $a[i]$ (resp. $b[i]$). Observe that Algorithm 1 updates a and b. We denote by d_a and d_b the degrees of these updated polynomials while we reserve n and m for the sizes of the initial values of a and b.

Algorithm 1 is implemented as a kernel which proceeds as follows. While $d_a \geq d_b$ holds, the polynomial a is replaced by $a - cX^d b$ where c is the leading coefficient of a divided by that of b and $d = d_a - d_b$. If this process makes a become zero, or makes d_a decrease by s while $d_a \geq d_b$ still holds, then the updated pair (a, b) is returned. If the condition $d_a \geq d_b$ becomes false before d_a could decrease by s, then the roles of a and b are exchanged, thus, a becomes the divisor.

Hence, each call to this kernel either computes the GCD of its input polynomials, or makes the sum of their degrees decrease at least by s. Since this may require to perform s division steps (using either a or b as divisor) each thread-block must compute the leading coefficients of a and b at each division step.

From the specifications of Algorithm 1, it follows that computing $\gcd(a, b)$ requires at most $\lceil \frac{n+m}{s} \rceil$ kernel calls. With the goal of optimizing the use of computing resources and sharpening the analysis of our multithreaded GCD computation, we observe that these kernel calls can be separated into two computational phases that we call *ping-ping* and *ping-pong*. During the *ping-ping* phase, the inequality $d_a \geq d_b$ remains true and thus this phase amounts to at most $\lceil \frac{n-m}{s} \rceil$ kernel calls. The subsequent kernel calls form the *ping-pong* phase during which the role of the divisor alternates between a and b; there are at most $\lceil \frac{2\,m}{s} \rceil$ kernel calls in that second phase.

Denoting by ℓ the number of threads per thread-block, we observe that each kernel call requires $\lceil \frac{m}{\ell} \rceil$ thread-blocks in *ping-ping* phase. During the *ping-pong* phase, for a kernel called on polynomials with current degrees d_a and d_b, the number of required thread-blocks becomes $\min(\lceil \frac{d_a+1}{\ell} \rceil, \lceil \frac{d_b+1}{\ell} \rceil)$.

After executing a kernel call in ping-ping phase, the s largest-degree coefficients of a have been set to zero and each thread block has updated ℓ other coefficients of a, meanwhile b is left unchanged. After executing a kernel call in ping-pong phase, s largest-degree coefficients among a or b ave been set to zero and each thread block has updated $2\,\ell$ other coefficients of a and b. See Figure 2 for an illustration of both scenarios.

To ensure that every kernel call (in either ping-ping or ping-pong phase) can perform (at most) s division steps correctly, each thread-block reads the s largest-degree coefficients from both a and b, as well as $\ell+s$ other consecutive coefficients from both a and b. Thereby, $4\,s + 2\,\ell$ coefficients must fit into the private memory of each streaming processor, hence, we have $4\,s + 2\,\ell \leq Z$.

The work, span and parallelism overhead are given [2] by $\mathrm{W}_s = 3\,m^2 + 6\,n\,m + 3\,s + \frac{3\,(5\,m\,s+4\,n\,s+14\,m+4\,n+3\,s^2+6\,s)}{8\,\ell}$, $\mathrm{S}_s = 3\,n + 3\,m$ and $\mathrm{O}_s = \frac{4\,m\,U\,(2\,n+m+s)}{s\,\ell}$, respectively.

To determine a value range for s that minimizes the parallelism overhead of our multithreaded algorithm, we choose $s = 1$ as starting point; let W_1, S_1 and O_1 the work, span, and parallelism overhead at $s = 1$. The work ratio $\mathrm{W}_1/\mathrm{W}_s$ is asymptotically equivalent to $\frac{(16\,\ell+8)\,n+(8\,\ell+19)\,m}{(16\,\ell+4\,s+4)\,n+(8\,\ell+5\,s+14)\,m}$ when n (and thus m) escapes to infinity. The span ratio $\mathrm{S}_1/\mathrm{S}_s$ is

[2] See the detailed analysis in the form of executable MAPLE worksheets of three applications: `http://www.csd.uwo.ca/~nxie6/projects/mcm/`.

Algorithm 1: OptGcdKer(a, b, s, da, db)

Input: $a, b \in \mathbb{K}[X]$, an integer $s \geq 1$ and da, db store the current degrees of a and b respectively.

Output: Either one of a, b was set to $\gcd(a, b)$ (and the other to 0), or da + db reduced at least by s.

Let Alc, Blc, A, B be local arrays of size $s, s, \ell + s, \ell + s$ respectively with coefficients in $\mathbb{K}$;

local integers $u = v = w = e = 0$;

j =blockID·blockDim + threadID; t = threadID;

/* copying from global memory */

if $t < s$ **then**

  Alc$[t]$ = a[da−t];

  Blc$[t]$ = b[db−t];

if $t \geq s$ **then**

  A$[t-s]$ = a[da−s blockID−t] B$[t-s]$ = b[db−s blockID−t];

/* computing next remainders */

for $(k = 0; k < s; k = k + 1)$ **do**

  if (da ≥ db **and** db ≥ 0) **then**

    if $(u + t < s)$ **and** $(v + t < s)$ **then**

      Alc$[u + t]$ -= Blc$[v + t]$·Alc$[u]$·Blc$[v]^{-1}$;

    if $(u + t \geq s)$ **and** $(v + t \geq s)$ **then**

      A$[w + t - s]$ -= B$[e + t - s]$·Alc$[u]$·Blc$[v]^{-1}$;

    if $t == 0$ **then**

      while Alc$[u] = 0$ **do**

        $u = u + 1$; $w = w + 1$; da = da−1;

  if (db ≥ da) **and** (da ≥ 0) **then**

    if $(u + t < s)$ **and** $(v + t < s)$ **then**

      Blc$[v + t]$ -= Alc$[u + t]$·Blc$[v]$·Alc$[u]^{-1}$;

    if $(u + t \geq s)$ **and** $(v + t \geq s)$ **then**

      B$[e + t - s]$ -= A$[w + t - s]$·Blc$[v]$·Alc$[u]^{-1}$;

    if $t == 0$ **then**

      while Blc$[v] = 0$ **do**

        $v = v + 1$; $e = e + 1$; db = db−1;

if $t \geq s$ **then**

  /* writing to global memory */

  a[da−s blockID−t] = A$[t-s]$;

  b[db−s blockID−t] = B$[t-s]$;

if $j ==$ min(da, db) **then**

  Update da, db with the new degrees of a and b;

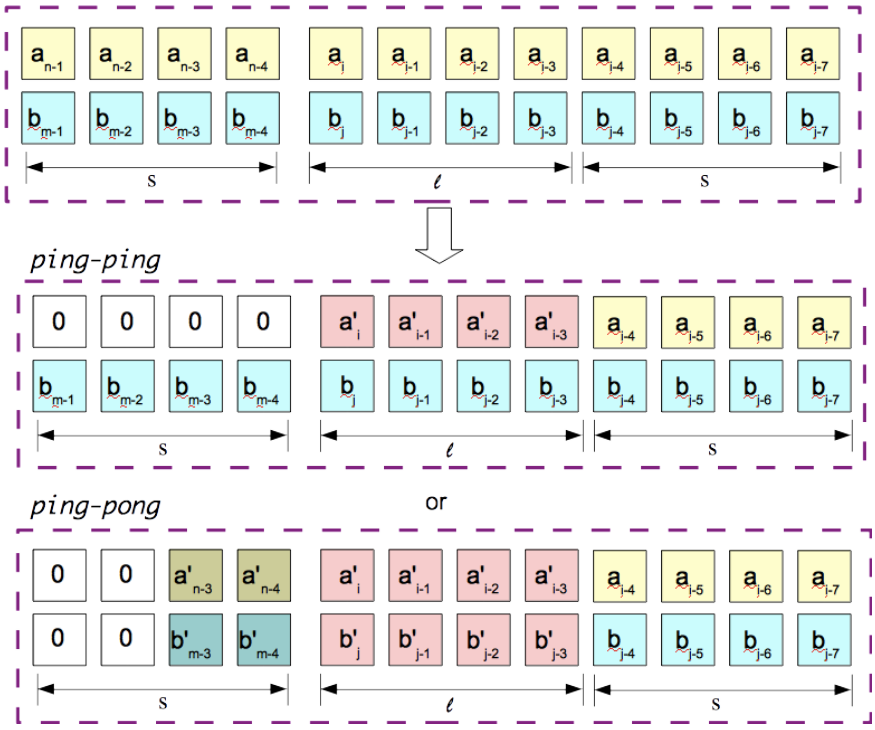

Figure 2: Illustration of reads and writes by a thread-block in either ping-ping or ping-pong phase.

1, and the parallelism overhead ratio O_1/O_s is $\frac{(2n+m+1)s}{2n+m+s}$. We observe that when $s \in \Theta(\ell)$, the work is increased by a constant factor only meanwhile the parallelism overhead will reduce by a factor in $\Theta(s)$.

Hence, choosing $s \in \Theta(\ell)$ seems a good choice. To verify this, we apply Corollary 1. One can easily check that the quantities characterizing the thread-block DAG of the computation are $N_s = \frac{2nm+m^2+ms}{2s\ell}$, $L_s = \frac{n+m}{s}$ and $C_s = 3s+8U$. Then, applying Corollary 1, we estimate the running time on $\Theta(\frac{m}{\ell})$ SMs as

$$T_s = \frac{4n+3m+s}{2s}(3s+8U).$$

Denoting by T_1 the estimated running time when $s = 1$, the running time ratio $R = T_1/T_s$ on $\Theta(\frac{m}{\ell})$ SMs is given by

$$R = \frac{(4n+3m+1)(3+8U)s}{(4n+3m+s)(3s+8U)}.$$

When n and m escape to infinity, the latter ratio asymptotically becomes $\frac{(3+8U)s}{3s+8U}$, which is greater than 1 if and only if $s > 1$. Thus, the algorithm with $s = \Theta(\ell)$ performs better than that with $s = 1$, Figure 3 shows the experimental results with $s = \ell = 256$ and $s = 1$ on a NVIDIA Kepler architecture, which confirms our theoretical analysis.

4 Fast Fourier Transform

Let f be a univariate polynomial over the prime field of characteristic p, namely $\mathbb{F}_p := \mathbb{Z}/p\mathbb{Z}$, where p is a prime number greater than 2. Let n be the smallest power of 2 such that the

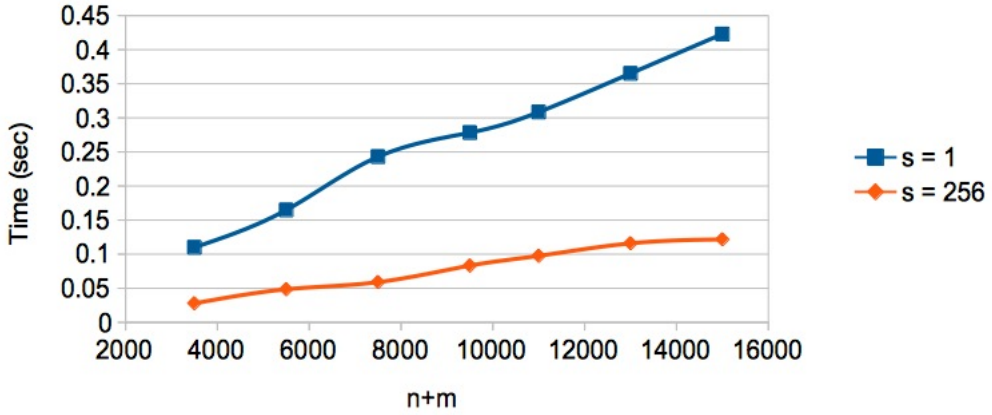

Figure 3: Running time on GeForce GTX 670 of our multithreaded Euclidean algorithm for univariate polynomials of sizes n and m over $\mathbb{Z}/p\mathbb{Z}$ where p is a 30-bit prime; the program parameter takes values $s = 1$ and $s = 256$.

degree of f is less than n, that is, $n = \min\{2^e \mid \deg(f) < 2^e \text{ and } e \in \mathbb{N}\}$. We assume that n divides $p-1$ which guarantees that the field $\mathbb{F}_p$ admits an n-th primitive root of unity. Hence, let $\omega \in \mathbb{F}_p$ such that $\omega^n = 1$ holds while for all $0 \le i < n$ we have $\omega^i \ne 1$. The n-point *Discrete Fourier Transform* (DFT) at ω is the linear map from the $\mathbb{F}_p$-vector space ${\mathbb{F}_p}^n$ to itself, defined by $x \longmapsto \mathrm{DFT}_n\, x$ with the n-th DFT matrix given by

$$\mathrm{DFT}_n = [\omega^{ij}]_{0 \le i,\, j < n}. \tag{4}$$

Since ω is an n-th primitive root of unity, this map is invertible and its inverse is $1/n$ times the DFT at ω', where ω' is the inverse of ω in $\mathbb{F}_p$. A *fast Fourier transform* (FFT) is an algorithm to compute the DFT and its inverse. Two of the most commonly used FFTs' are that of Cooley & Tukey [4] and that of Stockham [20]. Before reviewing and analyzing those algorithms on an MCM machine, we introduce a few notations. Given an input vector v of length $n\,m$, we define the *stride permutation* $L^{n\,m}_m$ as the permutation which maps the entry of v at position $i\,n+j$ to position $j\,m+i$, for $0 \le j \le (m-1)$, $0 \le i \le (n-1)$. We denote by I_x the identity matrix of order x. The symbols $\oplus$ and $\otimes$ are used for the *direct sum* and *Kronecker product* of matrices. We define the *twiddle matrix* $D_{x,y}$ as the diagonal matrix of order $x\,y$ given by $\bigoplus_{j=0}^{x-1} \mathrm{diag}(1, w^j, \ldots, w^{j\,(y-1)})$.

4.1 Cooley & Tukey algorithm

Let $k = \log_2(n)$ and fix $0 < r < k$. Define $m = 2^r$, hence m divides n. The algorithm of Cooley & Tukey is based on the following factorization[3] of the matrix DFT_n

$$\mathrm{DFT}_n = M_{k,r}\, M'_{k,r}\, M''_{k,r} \tag{5}$$

where $M_{k,r} = \prod_{i=0}^{k-r-1} (I_{2^i} \otimes \mathrm{DFT}_2 \otimes I_{2^{k-i-1}})(I_{2^i} \otimes D_{2,2^{k-i-1}})$, $M'_{k,r} = I_{2^{k-r}} \otimes \mathrm{DFT}_m$ and $M''_{k,r} = \prod_{i=k-r-1}^{0} I_{2^i} \otimes L_2^{2^{k-i}}$. Each of $M_{k,r}, M'_{k,r}, M''_{k,r}$ is a structured square matrix of order n representing a computational step of the algorithm.

[3]Matrix multiplications are not commutative. Throughout, the product $\prod_{i=1}^{s} M_i$ stands for $M_1 M_2 \cdots M_s$, while $\prod_{i=s}^{1} M_i$ means $M_s M_{s-1} \cdots M_1$.

As we shall see, the multiplication by the matrices $M''_{k,r}$ and $M_{k,r}$ are difficult to implement on a GPU-like architecture. Each factor of the matrix $M''_{k,r}$ is a Kronecker product of the form $I_{2^i} \otimes L_2^{2^{k-i}}$, thus a block diagonal matrix where each block has order 2^i and is a permutation matrix. One may assume that, for a kernel implementing the multiplication by $I_{2^i} \otimes L_2^{2^{k-i}}$, all thread blocks perform coalesced reads. But, for i small enough, several non-coalesced memory accesses to the global memory may occur when writing back within one thread-block. Turning our attention now to the multiplication by $M_{k,r}$, we notice that, when the matrix $I_{2^{i-1}} \otimes D_{2,2^{k-i}}$ operates on a sub-vector of length 2^{k-i+1}, the powers $\{1, \omega^{2^i}, (\omega^{2^i})^2, \ldots, (\omega^{2^i})^{2^{k-i}-1}\}$ need to be computed. A thread-block operating on an sub-vector of size x may need to compute x of those consecutive powers, which can be done in time $O(\log_2(x))$. Finally, multiplication by $M'_{k,r}$ causes no difficulty as long as m is large enough so as to avoid non-coalesced memory accesses. In our implementation $m = 16$ is appropriate.

Let ℓ be the number of threads per thread-block, then $M_{k,r}$ and $M''_{k,r}$ are computed by $\log_2(n) - \log_2(m)$ kernel calls, respectively, requiring $\frac{n}{\ell}$ thread-blocks per kernel, and $M'_{k,r}$ is computed by one kernel with $\frac{n}{m\,\ell}$ thread-blocks.

We compute the work, span, and parallelism overhead respectively as

$$\mathrm{W}_{ct} = n\,(34\,\log_2(n)\,\log_2(\ell) + 47\,\log_2(n) + 32\,m + 9 - 34\,\log_2(m)\,\log_2(\ell) - 47\,\log_2(m)),$$

$$\mathrm{S}_{ct} = 34\,\log_2(n)\,\log_2(\ell) + 47\,\log_2(n) + 32\,m\,\log_2(m) + 30\,m + 11 - 34\,\log_2(\ell)\,\log_2(m) - 79\,\log_2(m)$$

and

$$\mathrm{O}_{ct} = \frac{2\,n\,U}{\ell}\,(4\,\log_2(n) + \ell\,\log_2(\ell) + 1 - \log_2(\ell) - 4\,\log_2(m)).$$

To apply Corollary 1, one can easily check that those three quantities are $\mathrm{N}_{ct} = \frac{n}{m\,\ell}\,(2\,m\,\log_2(n) + 1 - 2\,m\,\log_2(m))$, $\mathrm{L}_{ct} = 2\,\log_2(n) - 2\,\log_2(m) + 1$ and $\mathrm{C}_{ct} = 2\,U\,\ell + 34\,\log_2(\ell) + 2\,U + 27$. Thus, we estimate that the running time on $\Theta(\frac{n}{\ell})$ streaming multiprocessors is

$$T_{ct} \quad = \quad (4\,\log_2(n) + 1 + \frac{1}{m} - 4\,\log_2(m))\,(2\,U\,\ell + 34\,\log_2(\ell) + 2\,U + 27)\,.$$

4.2 Stockham algorithm

The algorithm of Stockham is based on the following factorization of the matrix DFT_n

$$\mathrm{DFT}_n = \prod_{i=0}^{k-1} (\mathrm{DFT}_2 \otimes I_{2^{k-1}})(D_{2,2^{k-i-1}} \otimes I_{2^i})(L_2^{2^{k-i}} \otimes I_{2^i}),$$

where $k = \log_2(n)$ as before. For each $0 \leq i < k$, one performs three matrix-vector multiplications:

$$A_{k,i}\colon\ x \longmapsto \left(L_2^{2^{k-i}} \otimes I_{2^i}\right) x,$$

$$A'_{k,i}\colon\ x \longmapsto \left(D_{2,2^{k-i-1}} \otimes I_{2^i}\right) x,$$

$$A''_{k,i}\colon\ x \longmapsto \left(\mathrm{DFT}_2 \otimes I_{2^{k-1}}\right) x.$$

Thus, Stockham algorithm can be implemented as $\log_2(n)$ calls to a kernel performing successively the matrix-vector multiplications defined by $A_{k,i}$, $A'_{k,i}$ and $A''_{k,i}$. Each of the corresponding matrices has a structure permitting coalesced read/write memory accesses. See [16] for details.

Let ℓ be the number of threads per thread-block, thus each kernel requires $\frac{n}{\ell}$ thread-blocks. We compute the work, span, and parallelism overhead respectively as

$$\mathrm{W}_{sh} = n\,43\,\log_2(n) + \frac{n}{4\,\ell} + 12\,\ell + 1 - 30\,n,$$

$$\mathrm{S}_{sh} = 43\,\log_2(n) + 16\,\log_2(\ell) + 3$$

and

$$\mathrm{O}_{sh} = \frac{5\,n\,U\,\log_2(n)}{\ell} + \frac{5\,n\,U}{4\,\ell}.$$

Applying Corollary 1, the quantities characterizing the thread-block DAG are $\mathrm{N}_{sh} = \frac{n\,(8\,\log_2(n)-5)}{4\,\ell}$, $\mathrm{L}_{sh} = 3\,\log_2(n) + 1$, $\mathrm{C}_{sh} = 8\,\log_2(\ell) + 4\,U + 17$ and the running time estimate on $\Theta(\frac{n}{\ell})$ SMs is

$$T_{sh} = \log_2(n)\,(40\,\log_2(\ell) + 20\,U + 85) - 2\,\log_2(\ell) - U - \frac{17}{4}.$$

4.3 Comparison of running time estimates

The work ratio $\mathrm{W}_{ct}/\mathrm{W}_{sh}$ is asymptotically equivalent to

$$\frac{4\,n\,(47\,\log_2(n)\,\ell + 34\,\log_2(n)\,\ell\,\log_2(\ell))}{172\,n\,\log_2(n)\,\ell + n + 48\,\ell^2},$$

when n escapes to infinity. Since $\ell \in O(Z)$, the quantity ℓ is bounded over on a given machine. Thus, the work ratio is asymptotically equivalent to $\log_2(\ell)$ when n escapes to infinity, while the span ratio $\mathrm{S}_{ct}/\mathrm{S}_{sh}$ is asymptotically equivalent to

$$\frac{34\,\log_2(n)\,\log_2(\ell) + 47\,\log_2(n)}{43\,\log_2(n) + 16\,\log_2(\ell)},$$

which is also in $\Theta(\log_2(\ell))$. Next, we compute the parallelism overhead ratio, $\mathrm{O}_{ct}/\mathrm{O}_{sh}$, as

$$\frac{8\,n\,(4\,\log_2(n) + \ell\,\log_2(\ell) - \log_2(\ell) - 15)}{20\,n\,\log_2(n) + 5\,n - 4\,\ell}.$$

In other words, both the work and span of the algorithm of Cooley & Tukey are increased by $\Theta(\log_2(\ell))$ factor w.r.t their counterparts in Stockham algorithm. Applying Corollary 1, we obtain the running time ratio $R = T_{ct}/T_{sh}$ on $\Theta(\frac{n}{\ell})$ SMs as

$$R \sim \frac{\log_2(n)(2\,U\,\ell + 34\,\log_2(\ell) + 2\,U)}{5\,\log_2(n)\,(U + 2\,\log_2(\ell))},$$

when n escapes to infinity. This latter ratio is greater than 1 if and only if $\ell > 1$.

Hence, Stockham algorithm outperforms Cooley & Tukey algorithm on an MCM machine. Table 1 shows the experimental results comparing Cooley & Tukey and Stockham algorithms with $\ell = 128$ on a NVIDIA Kepler architecture, which confirms our theoretical analysis.

n	Cooley & Tukey	Stockham
2^{14}	0.583296	0.666496
2^{15}	0.826784	0.7624
2^{16}	1.19542	0.929632
2^{17}	2.07514	1.24928
2^{18}	4.66762	1.86458
2^{19}	9.11498	3.04365
2^{20}	16.8699	5.38781

Table 1: Running time (secs) of Cooley & Tukey and Stockham FFT algorithm with input size n on GeForce GTX 670.

5 Polynomial multiplication

Multithreaded algorithms for polynomial multiplication will be our third application of the MCM model in this paper. As in Section 3, we denote by a and b two univariate polynomials with coefficients in the prime field $\mathbb{F}_p$ and we write their degrees $\deg(a) = n-1$ and $\deg(b) = m-1$, for two positive integers $n \geq m$. We compute the product $f = a \times b$ in two ways: plain multiplication and FFT-based multiplication. We describe the plain multiplication approach in Section 5.1; we analyze it so as to tune a program parameter s and thus obtain an optimized algorithm. In Section 5.2, we analyze an FFT-based multiplication algorithm which uses Stockham FFT algorithm. Finally, we compare the plain and FFT-based multiplication algorithms in Section 5.3, via the MCM model and experimentally.

5.1 Plain multiplication

Our first multithreaded polynomial multiplication algorithm is based on the well-known *long multiplication*[4] and consists of two phases. During the *multiplication phase*, every coefficient of a is multiplied with every coefficient of b; the resulting coefficient products are accumulated in an auxiliary array M. Then, during the *addition phase*, these accumulated products are added together to form the polynomial f. The top level algorithm, shown in Algorithm 2, performs the multiplication phase via Algorithm 3, and the addition phase by repeated calls to Algorithm 4. We consider as program parameter the number $s > 0$ of coefficients that each thread writes back to the global memory at the end of each phase (multiplication or addition).

We denote by ℓ the number of threads per thread-block. In multiplication phase, each thread-block reads $s\,\ell + s - 1$ coefficients of a, 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 two-dimensional array M, whose format is $x \cdot y$, where $x = \frac{m}{s}$ and $y = n + s - 1$. This multiplication phase, illustrated by Figure 4, loads $s\,\ell + s - 1$ coefficients of a to guarantee the correctness of the results in M. Thereby, $2\,s\,\ell + 2\,s - 1$ coefficients must fit into the private memory, that is, we have $2\,s\,\ell + 2\,s - 1 \leq Z$, and the kernel requires $\frac{x\,y}{\ell\,s} = \frac{(n+s-1)\,m}{\ell\,s^2}$ thread-blocks.

In the addition phase, the x rows of the auxiliary array M are added pairwise in $\log_2(x)$ parallel steps. After each step, the number of rows in M is reduced by half, until we obtain

[4] `http://en.wikipedia.org/wiki/Multiplication_algorithm`

only one row that is, $f = a \times b$. Specifically, at a parallel step k ($0 \leq k < \log_2(x)$), the kernel requires $\frac{x\,y}{2^{k+1}\,\ell\,s} = \frac{(n+s-1)\,m}{2^{k+1}\,\ell\,s^2}$ thread-blocks, adding rows i and j (for $i < j$) shown in Figure 5, while each thread-block loads $s\,\ell$ elements of $M[i]$ and $M[j]$, respectively, and then adds $M[j]$ to $M[i]$.

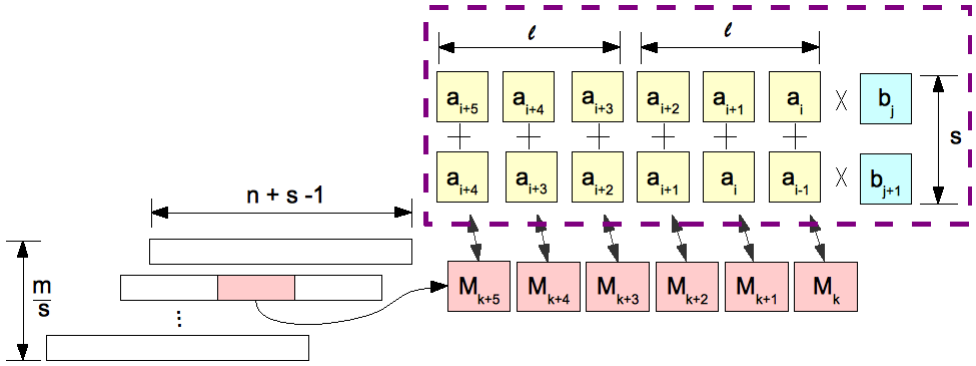

Figure 4: Multiplication phase: illustration of a thread-block reading coefficients from a,b and writing to the auxiliary array M.

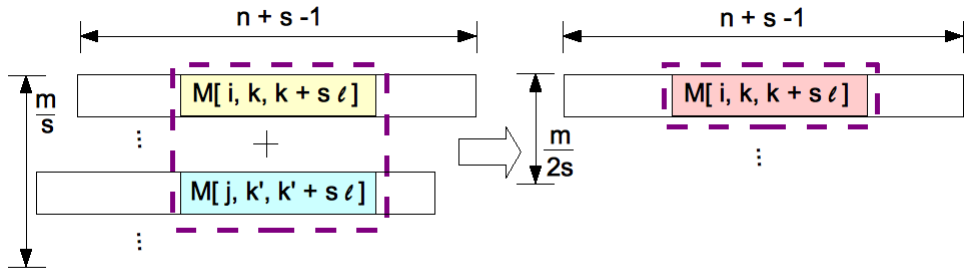

Figure 5: Addition phase: illustration of a thread-block reading and writing to the auxiliary array M.

We compute the work, span and parallelism overhead of the plain multiplication with an arbitrary s, respectively as $\mathrm{W}_s = (2\,m - \frac{1}{2})\,(n + s - 1)$, $\mathrm{S}_s = 2\,s^2 + s\,\log_2 \frac{m}{s} - s$ and $\mathrm{O}_s = \frac{(n+s-1)\,(5\,m\,s+2\,m-3\,s^2)\,U}{s^2\,\ell}$. We also obtain the quantities characterizing the thread-block DAG that are required in order to apply Corollary 1: $\mathrm{N}_s = \frac{(n+s-1)\,(2\,m-s)}{s^2\,\ell}$, $\mathrm{L}_s = \log_2(\frac{m}{s}) + 1$ and $\mathrm{C}_s = s\,(2\,s-1) + 2\,U\,(s+1)$.

We set $s = 1$ in the above two phases, and denote its work, span and parallelism overhead as W_1, S_1 and O_1 respectively. The work ratio $\mathrm{W}_1/\mathrm{W}_s = \frac{n}{n+s-1}$, is asymptotically constant as n escapes to infinity. The span ratio $\mathrm{S}_1/\mathrm{S}_s = \frac{\log_2(m)+1}{s\,(\log_2(m/s)+2\,s-1)}$ shows that S_s grows asymptotically with s. The parallelism overhead ratio is

$$\frac{\mathrm{O}_1}{\mathrm{O}_s} = \frac{n\,s^2\,(7\,m-3)}{(n+s-1)\,(5\,m\,s+2\,m-3\,s^2)}.$$

We observe that, as n and m escape to infinity, this latter ratio is asymptotically equivalent to s. Applying Corollary 1, the estimated running time on $\Theta(\frac{(n+s-1)\,m}{\ell\,s^2})$ SMs is

$$T_s = \left(\frac{2\,m-s}{m} + \log_2(\frac{m}{s}) + 1\right)\,(2\,U\,s + 2\,s^2 + 2\,U - s).$$

Algorithm 2: PlainMultiplicationGPU(a, b, s)

Input: $a, b \in \mathbb{F}_p[X]$ with $n := \deg(a) + 1$ and $m := deg(b) + 1$ and an integer $s \geq 1$.
Output: $f \in \mathbb{F}_p[X]$ and $f = a \times b$.

$y = n + s - 1$; $x = m/s$;
Let M be an array of size $x \cdot y$;
ℓ is the number of threads per block;
MulKer$\lll x \cdot y/(s \cdot \ell), \ell \ggg (a, b, M, n, m, s)$;
for $(i = 0; i < \log_2 x; i = i + 1)$ **do**
    AddKer$\lll x \cdot y/(2^{i+1} s \cdot \ell), \ell \ggg (M, f, y, s, x, i)$;
return f;

Algorithm 3: MulKer(a, b, M, n, m, s)

Input: $a, b, M \in \mathbb{F}_p[X]$ and an integer $s \geq 1$.

ℓ = blockDim; t = threadID; j =blockID$\cdot \ell + t$;
Let B and A be two local arrays of size s and $\ell \cdot s + s - 1$ respectively;
`/* copying from global */`
$i = s \cdot \lfloor s \cdot j/(n + s - 1) \rfloor + t$;
if $i < m$ **and** $t < s$ **then**
    $B[t] = b[i]$;
$i = s \cdot (j \mod \frac{n+s-1}{s})$;
for $(k = 0; k < s; k = k + 1)$ **do**
    if $i + k \cdot \ell + t < n$ **then**
        $A[k \cdot \ell + t] = a[i + k \cdot \ell + t]$;
if $i - s + t > 0$ **and** $t < s - 1$ **then**
    $A[\ell \cdot s + t] = a[i - s + t]$;
else if $t < s - 1$ **then**
    $A[\ell \cdot s + t] = 0$;
for $(e = 0; e < s; e = e + 1)$ **do**
    `/* accumulating products */`
    $h = 0$;
    for $(k = 0; k < s; k = k + 1)$ **do**
        $h \mathrel{+}= A[e \cdot \ell + k] \cdot B[k]$;
    `/* writing to global memory */`
    $M[s \cdot j + e] = h$;

Algorithm 4: AddKer(M, f, y, s, x, i)

Input: $M, f, \in \mathbb{F}_p[X]$ and y, s, x, i are positive integers.

```
j =blockID·blockDim + threadID;
h = s · j  mod y;
k = 2^i − 1 + 2^{i+1} ⌊s · j/y⌋;
if h < 2^i s then
    for (e = 0; e < s; e = e + 1) do
        f[k · s + h + e] += M[k · x + h + e];
else
    for (e = 0; e < s; e = e + 1) do
        M[(k + 2^i) · x + h − 2^i s + e] += M[k · x + h + e];
```

Let R be the ratio of the running time estimate between the algorithm with $s = 1$ and that for an arbitrary s, we obtain

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

which is asymptotically equivalent to $\frac{2\log_2(m)}{s \log_2(m/s)}$. This latter ratio is greater than 1 if and only if $s = 1$ or $s = 2$. In other words, increasing s makes the algorithm performance worse. In practice, shown on Figure 6, setting $s = 4$ (where $\ell = 256$) performs best, while with larger s, the running time becomes slower, which is coherent with our theoretical analysis.

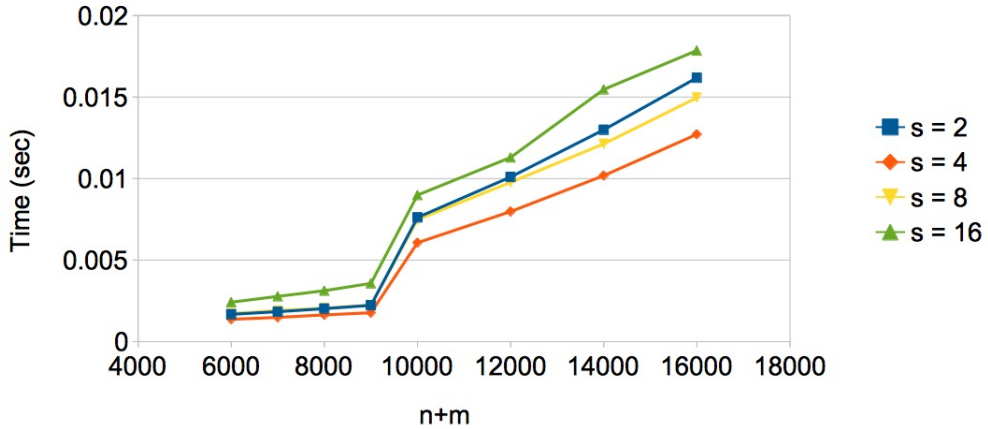

Figure 6: Running time of the plain polynomial multiplication algorithm with polynomials a ($\deg(a) = n - 1$) and b ($\deg(b) = m - 1$) and the parameter s on GeForce GTX 670.

5.2 FFT-based multiplication

Let ω be as n-th root primitive of unity w, as in Section 4. We assume that the successive powers $\{1, w, w^2, \ldots, w^{n-1}\}$ are pre-computed, say via a parallel prefix sum. The product $f = a \times b$ is computed as follows:

(i) FFT computations: $a' = \mathrm{DFT}_w(a)$ and $b' = \mathrm{DFT}_w(b)$;

(ii) Point-wise multiplication: $f' = a' \times b'$;

(iii) FFT computation: $f' = \mathrm{DFT}_{w^{-1}}(f')$;

(iv) Scale the vector: $f = \frac{1}{n} f'$ and **return** f;

A kernel for computing the Stockham FFT algorithm has been described in Section 4. Two other kernels perform respectively the point-wise multiplication and vector scaling.

Let ℓ be the number of threads per thread-block. The analysis of Stockham FFT algorithm is described in Section 4.3. To compute the point-wise multiplication, the kernel requires $\frac{n}{\ell}$ thread-blocks with each thread reading two data items and writing one data item back to the global memory. The final kernel, for vector scaling, also requires $\frac{n}{\ell}$ thread-blocks with each thread moving one data item among the global memory and private memories. Hence, data movement in each kernel call can be coalesced.

We compute the work, span, and parallelism overhead of the overall FFT-based polynomial multiplication as $\mathrm{W}_{fft} = 129\,n\,\log_2(n) - 94\,n$, $\mathrm{S}_{fft} = 129\,\log_2(n) - 94$ and $\mathrm{O}_{fft} = \frac{n\,U\,(15\,\log_2(n)-4)}{\ell}$. Applying Corollary 1, we check that we have $\mathrm{N}_{fft} = \frac{12\,n\,\log_2(n)-5}{2\,\ell}$, $\mathrm{L}_{fft} = 9\,\log_2(n) - 4$, and $\mathrm{C}_{fft} = 4\,U + 25$. Thus, the running time estimate on $\Theta(\frac{n}{\ell})$ SMs is

$$T_{fft} = (15\,\log_2(n) - \frac{13}{2})\,(4\,U + 25).$$

5.3 Comparison of running time estimates

Back to plain multiplication, using $s = 4$ obtained from experimental results and setting $m = n$, we compute $\mathrm{W}_{plain} = 2\,n^2 + \frac{11}{2}\,n - \frac{3}{2}$, $\mathrm{S}_{plain} = 4\,\log_2(n) + 20$, and $\mathrm{O}_{plain} = \frac{U\,(n+3)\,(11\,n-24)}{8\,\ell}$. We observe that $\mathrm{W}_{plain}/\mathrm{W}_{fft}$ is in $O(\frac{n}{\log_2(n)})$, $\mathrm{S}_{plain}/\mathrm{S}_{fft}$ is asymptotically constant, and $\mathrm{O}_{plain}/\mathrm{O}_{fft}$ is in $O(\frac{n}{\log_2(n)})$. Next, the estimated running time of the plain multiplication on $\Theta(\frac{n\,(n+3)}{16\,\ell})$ SMs is

$$T_{plain} = \left(\frac{2\,(n-2)}{n} + \log_2(n) - 1\right)\,(10\,U + 28).$$

The estimated running time ratio T_{plain}/T_{fft} is essentially constant when n escapes to infinity, although the plain multiplication performs more work and parallelism overhead.

However, the estimated running time of the plain multiplication on $\Theta(\frac{n}{\ell})$ SMs, in the case that we have limited resources, is

$$T'_{plain} = \left(\frac{(n+3)\,(n-2)}{8\,n} + \log_2(n) - 1\right)\,(10\,U + 28),$$

whereas the estimated running time of the FFT-based multiplication is also based on $\Theta(\frac{n}{\ell})$ SMs. Thus, when n escapes to infinity, the estimated running time ratio T'_{plain}/T_{fft} on $\Theta(\frac{n}{\ell})$ SMs is asymptotically equivalent to

$$\frac{5\,U\,(n + 8\,\log_2(n))}{240\,U\,\log_2(n)},$$

and thus in $\Theta(n)$. Hence, FFT-based multiplication outperforms the plain multiplication for n large enough.

Figure 7 shows the experimental results comparing the plain and FFT-based multiplication algorithms with $\ell = 256$ on a NVIDIA Kepler architecture. We observe that for relatively small n, plain multiplication performs better, but with n growing, the FFT-based multiplication becomes faster. Both phenomena were predicted by our theoretical analysis.

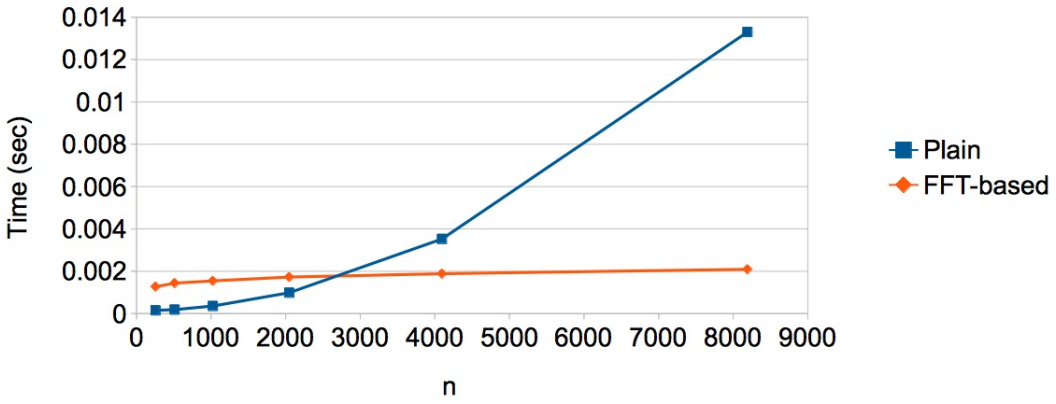

Figure 7: Running time of plain and FFT-based multiplication algorithms with input size n on GeForce GTX 670.

6 Conclusion

We have presented a model of multithreaded computation combining the fork-join and SIMD parallelisms, with an emphasis on estimating parallelism overheads of GPU programs. In practice, our model determines a trade-off among *work*, *span* and *parallelism overhead* by checking the estimated overall running time so as to (1) either tune a program parameter or, (2) compare different algorithms independently of the hardware details.

Several applications illustrate the effectiveness of our model. With the Euclidean algorithm and plain multiplication, we determine a range of values for a program parameter in order to optimize the corresponding algorithm in terms of parallelism overheads. With FFT algorithms and polynomial multiplication algorithms, our model successfully compare, in each case, two different algorithms to determine the trend of their running times when input data size grows. In all cases, experimentation validates the model analysis.

References

[1] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, 1990.

[2] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. Comput.*, 27(1):202–229, 1998.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[4] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.

[5] P. B. Gibbons. A more practical PRAM model. In *Proc. of SPAA*, pages 158–168. ACM, 1989.

[6] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. on Applied Mathematics*, 17(2):416–429, 1969.

[7] T. D. Han and T. S. Abdelrahman. Reducing branch divergence in GPU programs. In *Proc. of GPGPU-4*, pages 3:1–3:8. ACM, 2011.

[8] S. A. Haque. *Hardware Acceleration Technologies in Computer Algebra: Challenges and Impact*. PhD thesis, The University of Western Ontario, 2013.

[9] S. A. Haque, F. Mansouri, and M. Moreno Maza. On the parallelization of subproduct tree techniques targeting many-core architectures. In *Proc. of CASC 2014, LNCS 8660*, pages 171–185. Springer, 2014.

[10] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *Proc. of SPAA*, pages 145–156. ACM, 2010.

[11] Donald E. Knuth. *The Art of Computer Programming, Vol. II: Seminumerical Algorithms*. Addison-Wesley, 1969.

[12] W. Liu, W. Muller-Wittig, and B. Schmidt. Performance predictions for general-purpose computation on GPUs. In *Proc. of ICPP*, page 50. IEEE, 2007.

[13] L. Ma, K. Agrawal, and R. D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems*, 30:202–215, 2014.

[14] L. Ma and R. D Chamberlain. A performance model for memory bandwidth constrained applications on graphics engines. In *Proc. of ASAP*, pages 24–31. IEEE, 2012.

[15] L. Mirsky. A dual of Dilworth's decomposition theorem. *The American Math. Monthly*, 78(8):876–877, 1971.

[16] M. Moreno Maza and W. Pan. Fast polynomial arithmetic on a GPU. *J. of Physics: Conference Series*, 256, 2010.

[17] NVIDIA. NVIDIA next generation CUDA compute architecture: Kepler GK110, 2012.

[18] A. D. Robison. Composable parallel patterns with Intel Cilk Plus. *Computing in Science & Engineering*, 15(2):0066–71, 2013.

[19] J. Shin. Introducing control flow into vectorized code. In *Proc. of PACT*, pages 280–291. IEEE, 2007.

[20] T. G. Jr. Stockham. High-speed convolution and correlation. In *Proc. of AFIPS*, pages 229–233. ACM, 1966.

[21] L. J. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, 1984.