July 2015

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

Sardar Anisul HAQUE [a], Marc MORENO MAZA [a,b] and Ning XIE [a]
[a] *Department of Computer Science, University of Western Ontario, Canada*
[b] *ChongQing Institute for Green and Intelligent Technology, Chinese Academy of Sciences*

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 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.

Keywords. Model of computation, parallelism overhead, many-core architectures

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 solutions to a given algorithmic problem.

In the fork-join concurrency model [1], 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, namely the Graham-Brent theorem [1,2], which 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` [3]. 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 well-known PRAM (parallel random-access machine) model [4,5] has also been enhanced [6] 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 fails to capture all the features (and thus the impact) of data transfer between the SMs and the global memory of the device.

Ma, Agrawal and Chamberlain [7] introduce the TMM (Threaded Many-core Memory) model which retains many important characteristics of GPU-type architectures as machine parameters, like memory access width and hardware limit on the number of threads per core. In TMM analysis, the running time of an algorithm is estimated by choosing the maximum quantity among work, span and the amount of memory accesses. Such running time estimates depend on the machine parameters. Hong and Kim [8] present an analytical model to predict the execution time of an actual GPU program. No abstract machine is defined in this case. Instead, a few metrics are used to estimate the CPI (cycles per instruction) of the considered program.

Many works, such as [9,10], 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 with two objectives: (1) tuning program parameters to minimize parallelism overheads of algorithms targeting GPU-like architectures, and (2) comparing different algorithms independently of the targeted hardware device. In the design of this model, we insist on the following features:

1. *Two-level DAG programs.* Defined in Section 2, they capture the two levels of parallelism (fork-join and single instruction, multiple data) of heterogeneous programs (like a CilkPlus program using `#pragma simd` [11] or a CUDA program with the so-called dynamic parallelism [12]).
2. *Parallelism overhead.* We introduce this complexity measure in Section 2.3 with the objective of capturing communication and synchronization costs.
3. *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 of an MCM program on P streaming multiprocessors, see Theorem 1. However, as we shall see through a case study series, this machine parameter has no influence on the comparison of algorithms.

Our model extends both the fork-join concurrency 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 $\mathscr{P}$, that is, an algorithm expressed in the MCM model. Assume that a program parameter s (like the number of threads running on an SM) can be arbitrarily chosen within some range $\mathscr{S}$ while preserving the specifications of $\mathscr{P}$. Let s_0 be a particular value of s which corresponds to an instance $\mathscr{P}_0$ of $\mathscr{P}$, 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 $\mathrm{W}_{\mathscr{P}_0}/\mathrm{W}_{\mathscr{P}}$, $\mathrm{S}_{\mathscr{P}_0}/\mathrm{S}_{\mathscr{P}}$ and $\mathrm{O}_{\mathscr{P}_0}/\mathrm{O}_{\mathscr{P}}$. Assume that, when s varies within $\mathscr{S}$, the work ratio and span ratio stay within $O(s)$ (in fact, $\Theta(1)$ is often the case), but the ratio of the parallelism overhead reduces by a factor in $\Theta(s)$. Thereby, we determine a value $s_{\min} \in \mathscr{S}$ maximizing the parallelism overhead ratio. Next, we use our version of Graham-Brent

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

To evaluate the benefits of our model, we applied 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, the plain and FFT-based univariate polynomial multiplication algorithms. Other applications of our model appear in the PhD thesis [13] of the first Author as well as in [14].

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 the two fast Fourier transform algorithms and then the two univariate polynomial multiplication algorithms. In each case, work, span and parallelism overhead are evaluated so as to obtain running time estimates via our Graham-Brent theorem and then 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 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 and fixed size private memories with 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 concurrency model. A *kernel* is an SIMD (single instruction, multiple data) program capable of branches and decomposed into a 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

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

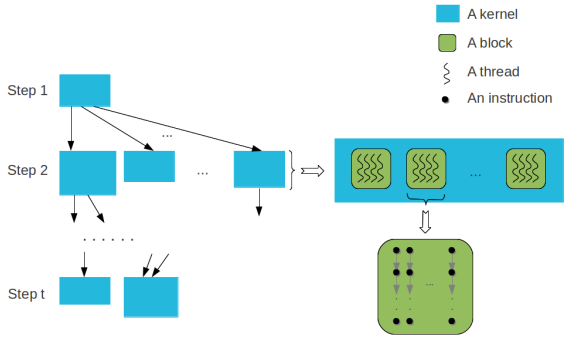

Figure 1. Overview of a many-core machine program

for each kernel the number of thread-blocks and the number of threads per thread-block. 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, an MCM program runs 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.

Toward 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; hence we have $U > 0$.
- 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 sets several characteristics and limitations 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 and the number of words in a data transfer between the global memory. The quantity $1/U$ is a throughput measure and has the following property. If α and β are the maximum numbers of words respectively read and written to the global memory by one thread of a thread-block B, and ℓ is 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:

$$T_D \leq (\alpha+\beta)U, \text{ if coalesced accesses occur, or } \ell(\alpha+\beta)U, \text{ otherwise.} \qquad (1)$$

On actual GPU devices, some hardware characteristics may reduce data transfer time, for instance, fast context switching between warps executed by a SM. Other hardware characteristics, like 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 lead us to 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 maximum 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 $T = T_A + T_D \leq \sigma V + \varepsilon(\alpha+\beta)U$, where ε is either 1 or ℓ. Instead of introducing this third machine parameter V, we let $V = 1$. Thus, U can be understood as the ratio of the time to transfer a machine word to the time to execute a local operation.

2.2. Many-core machine programs

Recall that each MCM program $\mathscr{P}$ is a DAG $(\mathscr{K}, \mathscr{E})$, called the *kernel DAG* of $\mathscr{P}$, where each node $K \in \mathscr{K}$ represents a kernel, and each edge $E \in \mathscr{E}$ records the fact that a kernel call must precede another kernel call. In other words, a kernel call can be executed once all its predecessors in the DAG $(\mathscr{K}, \mathscr{E})$ 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 [15]. 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 [16], 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 $\mathscr{P}$ is assumed to be known when $\mathscr{P}$ starts to execute, scheduling $\mathscr{P}$'s kernels onto the SMs can be done in time $O(\Gamma)$ where Γ is the total length of $\mathscr{P}$'s kernel code. Thus, we neglect those costs in comparison to the costs of data transfer between SMs' private memories and the global memory. Extending MCM machines to program DAGs unfolding dynamically at run time is work in progress.

Thread-block DAG. Since each kernel of the program $\mathscr{P}$ decomposes into finitely many thread-blocks, we map $\mathscr{P}$ to a second graph, called the *thread-block DAG* of $\mathscr{P}$, whose vertex set $\mathscr{B}(\mathscr{P})$ consists of all thread-blocks of the kernels of $\mathscr{P}$ 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 $\mathscr{P}$. This second graph defines two important quantities:

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

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

2.3. Complexity measures for the many-core machine model

Consider an MCM program $\mathscr{P}$ given by its kernel DAG $(\mathscr{K},\mathscr{E})$. Let $K \in \mathscr{K}$ be any kernel of $\mathscr{P}$ 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. As before, 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

$$(\alpha+\beta)U, \text{if memory accesses can be coalesced or } \ell(\alpha+\beta)U, \text{otherwise.} \qquad (2)$$

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 $\mathscr{P}$. The *work* $\mathrm{W}(\mathscr{P})$ of $\mathscr{P}$ is defined as the total work of all its kernels. Regarding the graph $(\mathscr{K},\mathscr{E})$ as a weighted-vertex graph, where the weight of a vertex $K \in \mathscr{K}$ is its span $\mathrm{S}(K)$, we define the weight $\mathrm{S}(\gamma)$ of any path γ from the first executing kernel to a terminal kernel (that is, a kernel with no successors in $\mathscr{P}$) as $\mathrm{S}(\gamma) = \sum_{K\in\gamma} \mathrm{S}(K)$. Then, we define the *span* $\mathrm{S}(\mathscr{P})$ of $\mathscr{P}$ as the longest path, counting the weight (span) of each vertex (kernel), in the kernel DAG. Finally, we define the *overhead* $\mathrm{O}(\mathscr{P})$ of the program $\mathscr{P}$ as the total overhead of all its kernels. Observe that, according to Mirsky's theorem [17], the number π of parallel steps in $\mathscr{P}$ (which form a partition of $\mathscr{K}$ into anti-chains in the DAG $(\mathscr{K},\mathscr{E})$ regarded as a partially ordered set) is greater or equal to the maximum length of a path in $(\mathscr{K},\mathscr{E})$ from the first executing kernel to a terminal kernel.

2.4. A Graham-Brent theorem with parallelism overhead

Theorem 1 *The running time T_P of the program $\mathscr{P}$ executed on P SMs satisfies the inequality: $T_{\mathsf{P}} \leq (\mathrm{N}(\mathscr{P})/\mathsf{P} + \mathrm{L}(\mathscr{P}))\mathrm{C}(\mathscr{P})$, where $\mathrm{C}(\mathscr{P}) = \max_{B\in\mathscr{B}(\mathscr{P})}(\mathrm{S}(B)+\mathrm{O}(B))$.*

The proof is similar to that of the original result [1,2]. while 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 $\mathscr{P}$. Then the running time $T_{\mathscr{P}}$ of the program $\mathscr{P}$ satisfies:*

$$T_{\mathscr{P}} \leq (\mathrm{N}(\mathscr{P})/\mathsf{K} + \mathrm{L}(\mathscr{P}))\mathrm{C}(\mathscr{P}). \qquad (3)$$

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 $\mathscr{P}$. 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 the 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 over a the finite field $\mathbb{Z}/p\mathbb{Z}$, where p is a prime number. Our approach is based on the Euclidean algorithm, that the reader can review in Chapter 4 in [18]. Given a positive integer s, we proceed by repeatedly calling a subroutine which takes as input a pair (a,b) of polynomials in $\mathbb{Z}/p\mathbb{Z}[X]$, with $\deg(a) \geq \deg(b) > 0$, and returns another pair (a', b') of polynomials in $\mathbb{Z}/p\mathbb{Z}[X]$, such that $\gcd(a,b) = \gcd(a', b')$, and either $b' = 0$ (in which case we have $\gcd(a,b) = a'$), or we have $\deg(a') + \deg(b') \leq \deg(a) + \deg(b) - s$. Details, including pseudo-code, can be found in the long version of this paper available at `http://cumodp.org/hmx2015-draft.pdf`.

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 the degree of the input polynomials a and b (in dense representation) satisfies $\deg(a) = n-1$ and $\deg(b) = m-1$, assuming $n \geq m$.

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

To determine a value range for s that minimizes the parallelism overhead of our multithreaded algorithm, we choose $s = 1$ as the 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 $\frac{(16\ell+8)n+(8\ell+19)m}{(16\ell+4s+4)n+(8\ell+5s+14)m}$ when m (and thus n) escapes to infinity. The span ratio $\mathrm{S}_1/\mathrm{S}_s$ is 1, and the parallelism overhead ratio $\mathrm{O}_1/\mathrm{O}_s$ 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 $\mathrm{N}_s = \frac{2nm+m^2+ms}{2s\ell}$, $\mathrm{L}_s = \frac{n+m}{s}$ and $\mathrm{C}_s = 3s+8U$. 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 2 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 p be a prime number greater than 2 and let f be a vector over the prime field $\mathbb{F}_p := \mathbb{Z}/p\mathbb{Z}$. Let n be the smallest power of 2 such that the length 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

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

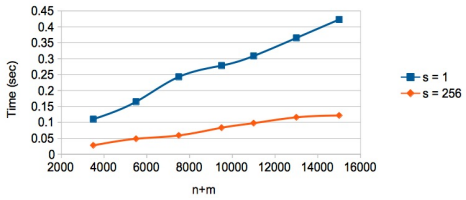

Figure 2. Running time on GeForce GTX 670 of our multithreaded Euclidean algorithm for univariate polynomials of sizes n and m over $\mathbb{Z}/p\mathbb{Z}$ where p is a 30-bit prime; the program parameter s takes 1 and 256.

$\omega^n = 1$ holds while for all $0 \leq i < n$, we have $\omega^i \neq 1$. The n-point *Discrete Fourier Transform* (DFT, for short) 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 $\mathrm{DFT}_n = [\omega^{ij}]_{0 \leq i,\, j < n}$. A *fast Fourier transform* (FFT, for short) is an algorithm to compute the DFT. Two of the most commonly used FFTs' are that of Cooley & Tukey [19] and that of Stockham [20]. Details, including a review of those algorithms, can be found in the long version of this paper available at `http://cumodp.org/hmx2015-draft.pdf`. Each of these algorithms is based on a factorization of the matrix DFT_n, which translates into $\log_2(n)$ calls to a kernel performing successively three matrix-vector multiplications. In the case of Stockham's factorization, each of the corresponding matrices has a structure permitting coalesced read/write memory accesses. Unfortunately, this is not always true for the factorization of Cooley & Tukey. As we shall see, the MCM model can quantify this negative feature of this latter algorithm, thus yielding an analytical explanation to a fact which, up to our knowledge, had never measured in such precise way in the literature.

Estimates for the work, span, and parallelism overhead of each algorithm appear in `http://cumodp.org/hmx2015-draft.pdf`. In what follows, W_{ct}, S_{ct} and O_{ct} refer to the work, span, and parallelism overhead of the algorithm of Cooley & Tukey. Similarly, W_{sh}, S_{sh} and O_{sh} stand for the work, span, and parallelism overhead of Stockham's.

The work ratio $\mathrm{W}_{ct}/\mathrm{W}_{sh}$ is asymptotically equivalent to $\frac{4n(47\log_2(n)\ell+34\log_2(n)\ell\log_2(\ell))}{172n\log_2(n)\ell+n+48\ell^2}$, 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 in $\Theta(\log_2(\ell))$ 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 $\frac{8n(4\log_2(n)+\ell\log_2(\ell)-\log_2(\ell)-15)}{20n\log_2(n)+5n-4\ell}$. 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 $R \sim \frac{\log_2(n)(2U\ell+34\log_2(\ell)+2U)}{5\log_2(n)(U+2\log_2(\ell))}$, 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 both algorithms with $\ell = 128$ on a NVIDIA Kepler architecture, which confirms our theoretical analysis.

n	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
Cooley & Tukey (secs)	0.583	0.826	1.19	2.07	4.66	9.11	16.8
Stockham (secs)	0.666	0.762	0.929	1.24	1.86	3.04	5.38

Table 1. Running time 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.

Our multithreaded algorithm for plain multiplication was introduced in [21] and is reviewed with details in `http://cumodp.org/hmx2015-draft.pdf`. This algorithm depends on a program parameter $s > 0$ which is the number 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.

We see $s = 1$ as our initial algorithm; we 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 $\mathrm{S}_1/\mathrm{S}_s = \frac{\log_2(m)+1}{s(\log_2(m/s)+2s-1)}$ shows that S_s grows asymptotically with s. The parallelism overhead ratio is $\mathrm{O}_1/\mathrm{O}_s = \frac{ns^2(7m-3)}{(n+s-1)(5ms+2m-3s^2)}$. We observe that, as n and m escape to infinity, this latter ratio is asymptotically in $\Theta(s)$. Applying Corollary 1, the estimated running time on $\Theta(\frac{(n+s-1)m}{\ell s^2})$ SMs is $T_s = \left(\frac{2m-s}{m} + \log_2(\frac{m}{s}) + 1\right)(2Us + 2s^2 + 2U - s)$. One checks that the running time estimate ratio is asymptotically equivalent to $\frac{2U\log_2(m)}{s(s+U)\log_2(m/s)}$. This latter is smaller than 1 for $s > 1$. Hence, increasing s makes the algorithm performance worse. In practice, as shown on Figure 3, setting $s = 4$ (where $\ell = 256$) performs best, while a larger s increases the running time, which is coherent with our theoretical analysis.

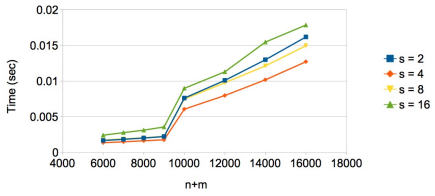

Figure 3. Running time of plain polynomial multiplication algorithm with dense polynomials of sizes n, m and parameter s on GeForce GTX 670.

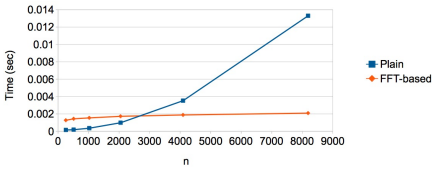

Figure 4. Running time of plain and FFT-based multiplication algorithms with input size n on GeForce GTX 670.

We consider now an alternative polynomial multiplication, based on FFT, see for instance [22]. Let ℓ be the number of threads per thread-block. Based on the analysis of Stockham FFT algorithm, we obtain the work, span, and parallelism overhead of the overall FFT-based polynomial multiplication as $\mathrm{W}_{fft} = 129n\log_2(n) - 94n$, $\mathrm{S}_{fft} = 129\log_2(n) - 94$ and $\mathrm{O}_{fft} = \frac{nU(15\log_2(n)-4)}{\ell}$. Applying Corollary 1, the running time estimate on $\Theta(\frac{n}{\ell})$ SMs is $T_{fft} = (15\log_2(n) - \frac{13}{2})(4U + 25)$.

Back to plain multiplication, using $s = 4$ obtained from experimental results and setting $m = n$, we observe that the estimated running time ratio T_s/T_{fft} is essentially constant on $\Theta(\frac{n^2}{\ell})$ SMs[3] when n escapes to infinity, although the plain multiplication performs more work and parallelism overhead.

[3] This is the amount of SMs required in the above estimates for the plain multiplication.

However, the estimated running time of the plain multiplication on $\Theta(\frac{n}{\ell})$ SMs becomes $T'_{plain} = \left(\frac{(n+3)(n-2)}{8n} + \log_2(n) - 1\right)(10U + 28)$, that is, in a context of limited resource (namely SMs) w.r.t. the previous estimate. Since the running time estimate for FFT-based multiplication is also based on $\Theta(\frac{n}{\ell})$ SMs, we observe that, when n escapes to infinity, the ratio T'_{plain}/T_{fft} on $\Theta(\frac{n}{\ell})$ SMs is asymptotically equivalent to $\frac{5U(n+8\log_2(n))}{240U\log_2(n)}$, thus in $\Theta(n)$. Therefore, FFT-based multiplication outperforms plain multiplication for n large enough, when resources are limited. Figure 4 shows coherent experimental results with $\ell = 256$. In conclusion, the MCM model can take available resources into account when comparing two algorithms.

References

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

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

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

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

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

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

[7] 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.

[8] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, June 2009.

[9] 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.

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

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

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

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

[14] 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.

[15] 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.

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

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

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

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

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

[21] S. A. Haque and M. Moreno Maza. Plain polynomial arithmetic on GPU. In *J. of Physics: Conf. Series*, volume 385, page 12014. IOP Publishing, 2012.

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