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

Sardar Anisul Haque  Marc Moreno Maza  Ning Xie

University of Western Ontario, Canada

IBM CASCON, November 4, 2014

# Optimize algorithms targeting GPU-like many-core devices

## Background

- Given a CUDA code, an experimented programmer may attempt well-known strategies to improve the code performance in terms of arithmetic intensity and memory bandwidth.
- Given a CUDA-like algorithm, one would like to derive code for which much of this optimization process has been lifted at the design level, i.e. before the code is written.

## Methodology

We need a model of computation which

- captures the computer hardware characteristics that have a dominant impact on program performance.
- combines its complexity measures (work, span) so as to determine the *best* algorithm among different possible algorithmic solutions to a given problem.

# Challenges in designing a model of computation

## Theoretical aspects

- GPU-like architectures introduces many machine parameters (like memory sizes, number of cores), and too many could lead to intractable calculations.
- GPU-like code depends also on program parameters (like number of threads per thread-block) which specify how the work is divided among the computing resources.

## Practical aspects

- One wants to avoid answers like: *Algorithm 1 is better than Algorithm 2* providing that the machine parameters satisfy a system of constraints.
- We prefer analysis results independent of machine parameters.
- Moreover, this should be achieved by selecting program parameters in appropriate ranges.

# Overview

- We present a model of multithreaded computation with an emphasis on estimating parallelism overheads of programs written for modern many-core architectures.
- 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 [1] and the results of their experimental comparison are coherent with the theoretical analysis based on our model.

[1] Publicly available written in CUDA from `http://www.cumodp.org/`

# Plan

1. Models of computation
   - Fork-join model
   - Parallel random access machine (PRAM) model
   - Threaded many-core memory (TMM) model
2. A many-core machine (MCM) model
3. Experimental validation
4. Concluding remarks

# Fork-join model

This model has become popular with the development of the concurrency platform CilkPlus, targeting multi-core architectures.

- The *work* $T_1$ is the total time to execute the entire program on one processor.
- The *span* $T_\infty$ is the longest time to execute along any path in the DAG.
- We recall that the Graham-Brent theorem states that the running time $T_P$ on P processors satisfies $T_P \leq T_1/P + T_\infty$. A refinement of this theorem captures scheduling and synchronization costs, that is, $T_P \leq T_1/P + 2\delta\widehat{T_\infty}$, where $\delta$ is a constant and $\widehat{T_\infty}$ is the burdened span.

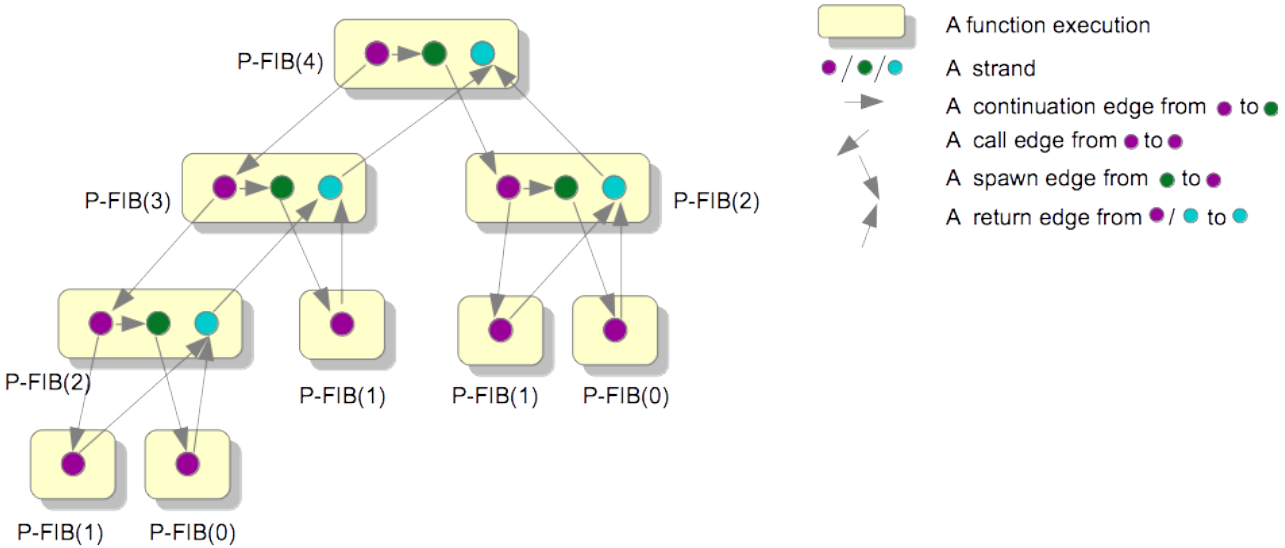



Figure: An example of computation DAG: 4-th Fibonacci number

# Parallel random access machine (PRAM) model

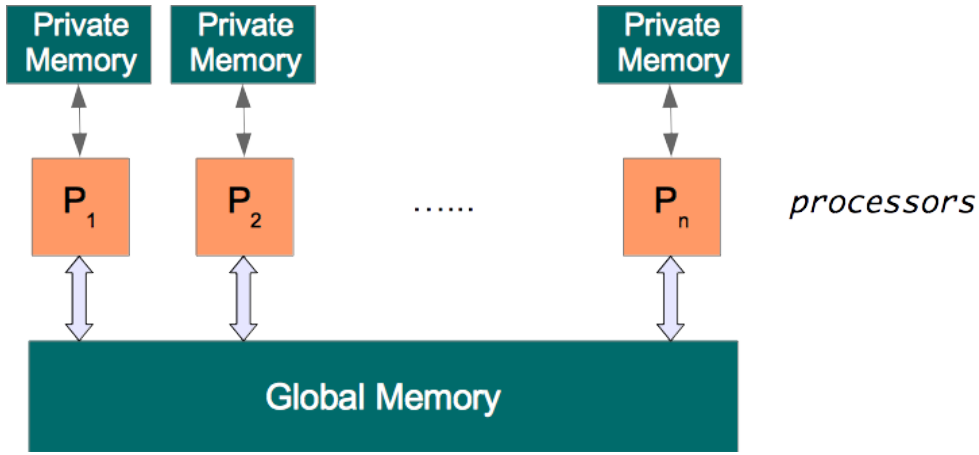



Figure: Abstract machine of PRAM model

- Instructions on a processor execute in a 3-phase cycle: read-compute-write.
- Processors access to the global memory in a unit time (unless an access conflict occurs).
- These strategies deal with read/write conflicts to the same global memory cell: EREW, CREW and CRCW (exclusive or concurrent).
- A refinement of PRAM integrates communication delay into the computation time.

# Hybrid CPU-GPU system

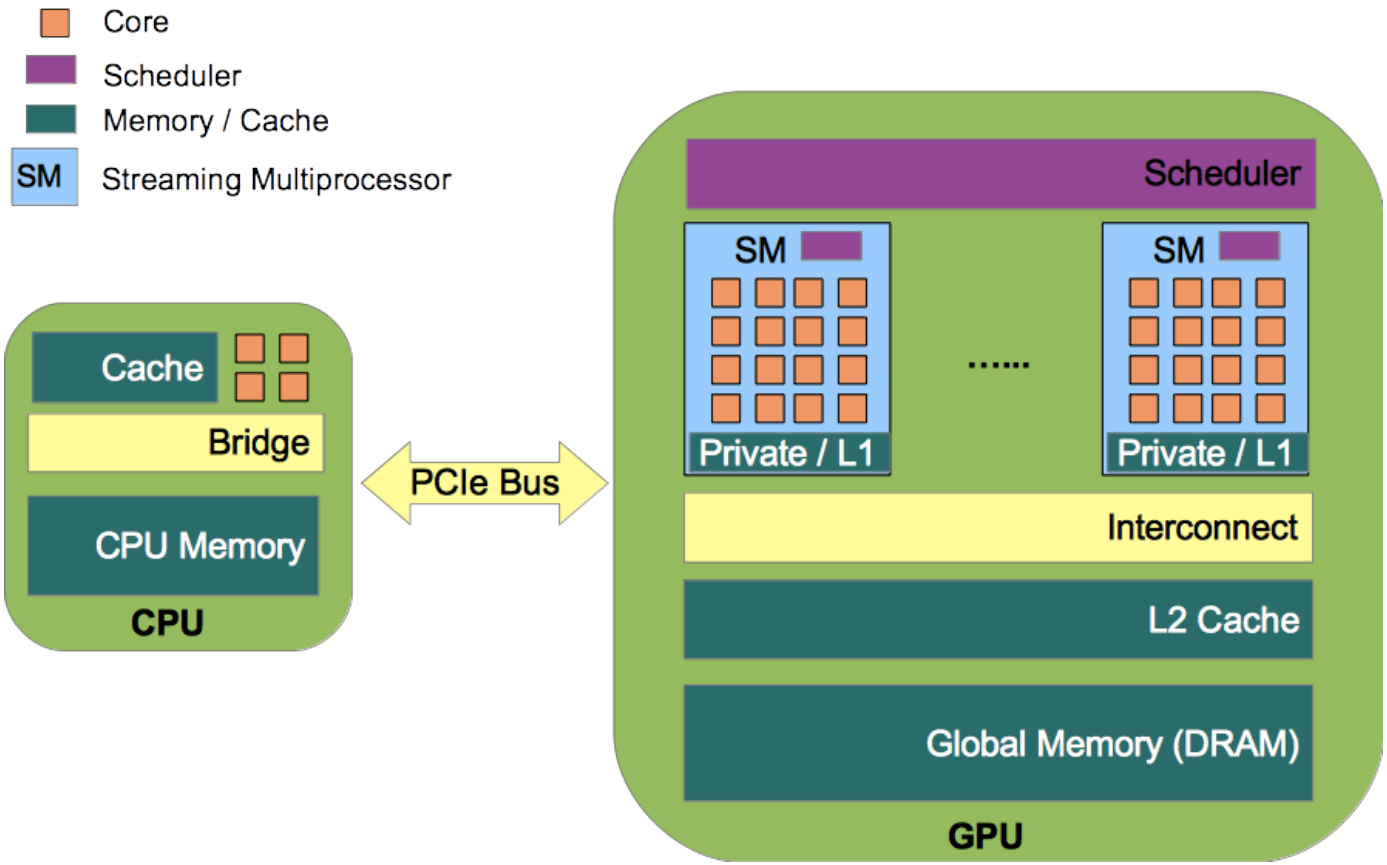



Figure: Overview of a hybrid CPU-GPU system

# Threaded many-core memory (TMM) model

Ma, Agrawal and Chamberlain introduce the TMM model which retains many important characteristics of GPU-type architectures.

| | Description |
|---|---|
| L | Time for a global memory access |
| P | Number of processors (cores) |
| C | Memory access width |
| Z | Size of fast private memory per core group |
| Q | Number of cores per core group |
| X | Hardware limit on number of threads per core |

Table: Machine parameters of the TMM model

- In TMM analysis, the running time of algorithm is estimated by choosing the maximum quantity among the work, span and amount of memory accesses. No Graham-Brent theorem-like is provided.
- Such running time estimates depend on the machine parameters.

# Plan

1. Models of computation
2. A many-core machine (MCM) model
   - Characteristics
   - Complexity measures
3. Experimental validation
4. Concluding remarks

# A many-core machine (MCM) model

We propose a many-core machine (MCM) model which aims at

- tuning program parameters to minimize parallelism overheads of algorithms targeting GPU-like architectures as well as
- comparing different algorithms independently of the value of machine parameters of the targeted hardware device.

In the design of this model, we insist on the following features:

- Two-level DAG programs
- Parallelism overhead
- A Graham-Brent theorem

# Characteristics of the abstract many-core machines

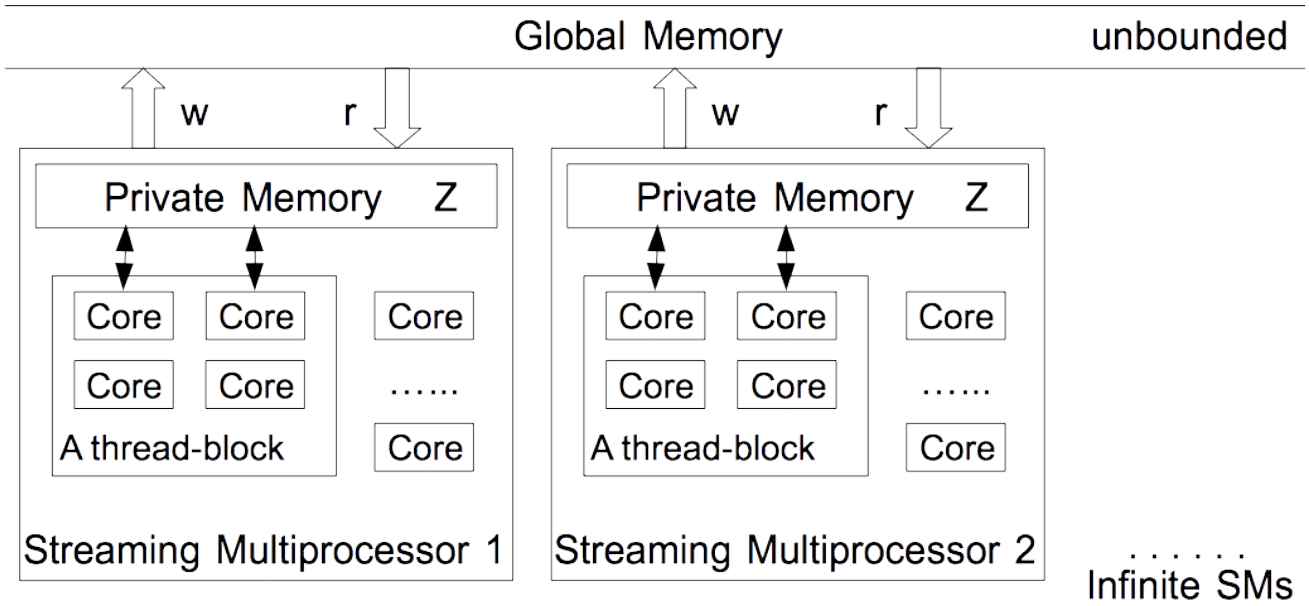



Figure: A many-core machine

- It has a global memory with high latency and low throughput while private memories have low latency and high throughput

# Characteristics of the abstract many-core machines

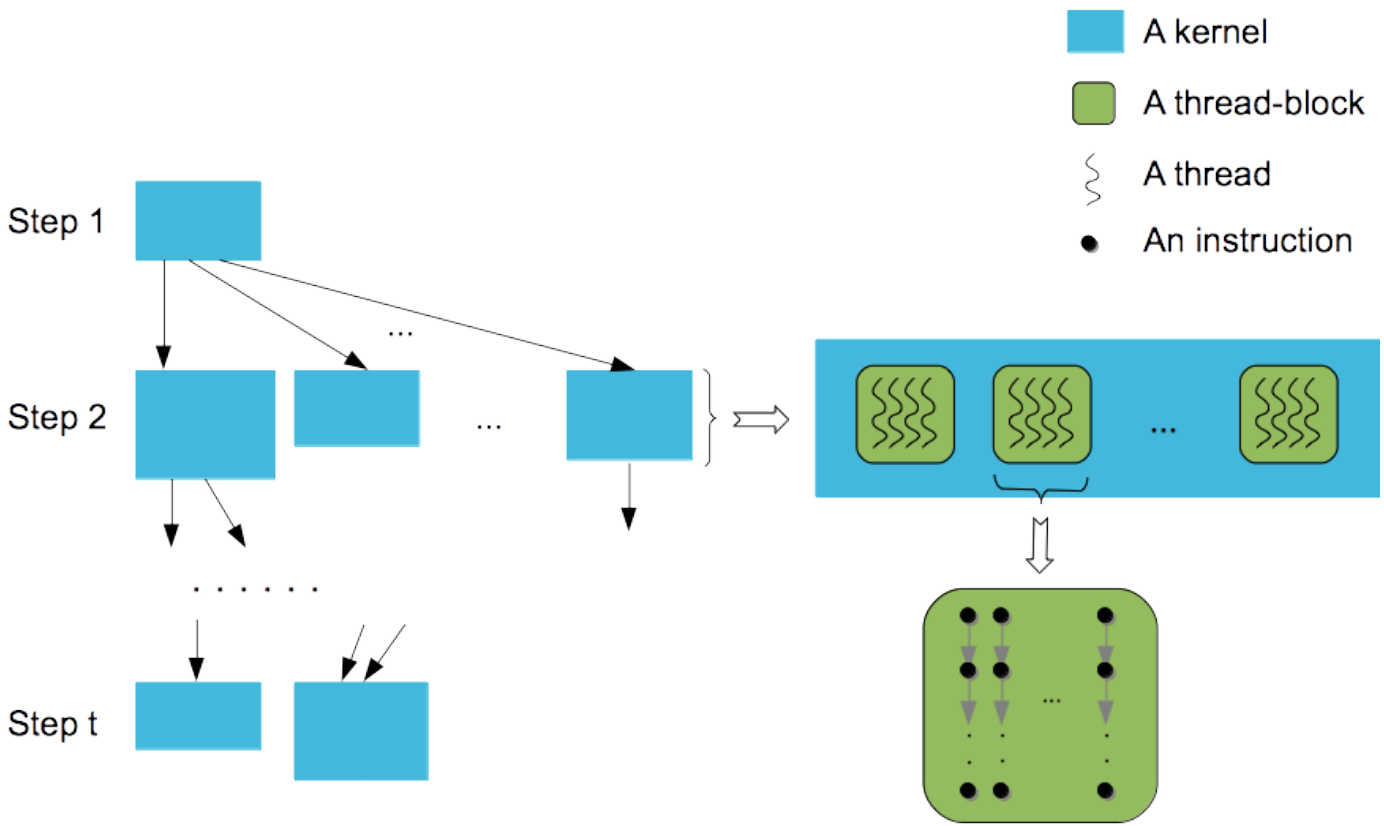



Figure: Overview of a many-core machine program

# Characteristics of the abstract many-core machines

## Synchronization costs

- It follows that MCM kernel code needs no synchronization statement.
- Consequently, the only form of synchronization taking place among the threads executing a given thread-block is that implied by code divergence.
- An MCM machine handles code divergence by eliminating the corresponding conditional branches via code replication, and the corresponding cost will be captured by the complexity measures (work, span and parallelism overhead) of the MCM model.

# Machine parameters of the abstract many-core machines

## Z: Private memory size of any SM

- It sets up an upper bound on several program parameters, for instance, 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.

## U: Data transfer time

- Time (expressed in clock cycles) to transfer one machine word between the global memory and the private memory of any SM, that is, $U > 0$.
- As an abstract machine, the MCM aims at capturing either the best or the worst scenario for data transfer time of a thread-block, that is,

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

where $\alpha$ and $\beta$ are the numbers of words respectively read and written to the global memory by one thread of a thread-block $B$ and $\ell$ be the number of threads per thread-block.

# Complexity measures for the many-core machine model

For any kernel $\mathcal{K}$ of an MCM program,

- **work** $W(\mathcal{K})$ is the total number of local operations of all its threads;
- **span** $S(\mathcal{K})$ is the maximum number of local operations of one thread;
- **parallelism overhead** $O(\mathcal{K})$ is the total data transfer time among all its thread-blocks.

For the entire program $\mathcal{P}$,

- **work** $W(\mathcal{P})$ is the total work of all its kernels;
- **span** $S(\mathcal{P})$ is the longest path, counting the weight (span) of each vertex (kernel), in the kernel DAG;
- **parallelism overhead** $O(\mathcal{P})$ is the total parallelism overhead of all its kernels.

# Characteristic quantities of the thread-block DAG

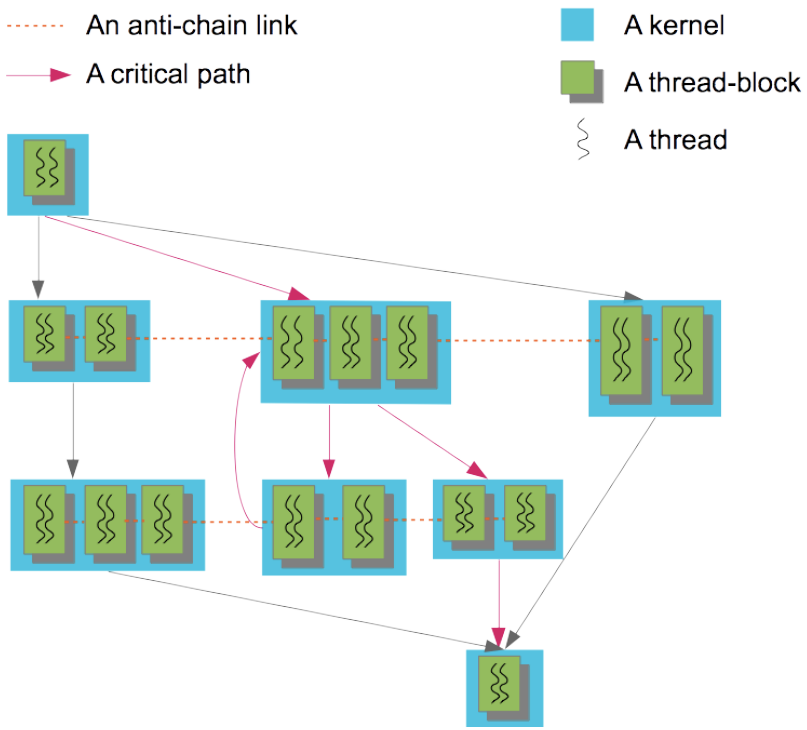



Figure: Thread-block DAG of a many-core machine program

$\mathrm{N}(\mathcal{P})$: 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 $\mathcal{P}$.

# Complexity measures for the many-core machine model

## Theorem (A Graham-Brent theorem with parallelism overhead)

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

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

*where* $\mathrm{C}(\mathcal{P})$ *is the maximum running time of local operations (including read/write requests) and data transfer by one thread-block.*

## Corollary

*Let* $\mathsf{K}$ *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:*

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

# Plan


1. Models of computation
2. A many-core machine (MCM) model
3. Experimental validation
4. Concluding remarks

# Tuning program parameter with MCM model

For an MCM program $\mathcal{P}$ depending on program parameter $s$ varying in a range $\mathcal{S}$.

- Let $s_0$ be an "initial" value of $s$ corresponding to an instance $\mathcal{P}_0$ of $\mathcal{P}$.
- Assume the work ratio $W_{s_0}/W_s$ remains essentially constant meanwhile the parallelism overhead $O_s$ varies more substantially, say $O_{s_0}/O_s \in \Theta(s - s_0)$.
- Then, we determine a value $s_{\min} \in \mathcal{S}$ maximizing the ratio $O_{s_0}/O_s$.
- Next, we use our version of Graham-Brent theorem to confirm that the upper bound for the running time of $\mathcal{P}(s_{\min})$ is less than that of $\mathcal{P}(s_0)$.

# Plain (or long) univariate polynomial multiplication

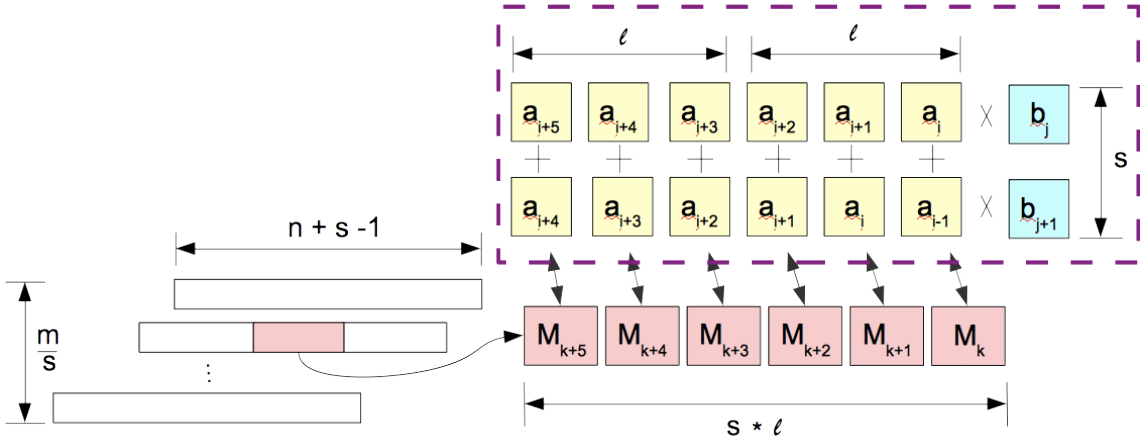
We denote by $a$ and $b$ two univariate polynomials (or vectors) with sizes $n \geq m$. We compute the product $f = a \times b$.

$$
\begin{array}{cccccccc}
 & & a_3 & a_2 & a_1 & a_0 & \times & b_0 \\
 & & + & + & + & & & \\
 & a_3 & a_2 & a_1 & a_0 & & \times & b_1 \\
 & + & + & + & & & & \\
a_3 & a_2 & a_1 & a_0 & & & \times & b_2 \\
--- & --- & --- & --- & --- & --- & & \\
f_5 & f_4 & f_3 & f_2 & f_1 & f_0 & &
\end{array}
$$

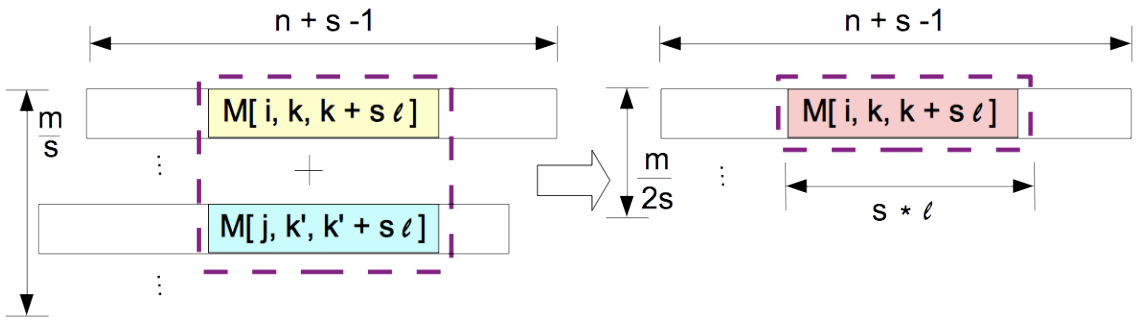
- This long multiplication process has two phases.
- Matrix multiplication follows a similar pattern.

# Plain univariate polynomial multiplication

*Multiplication phase*: every coefficient of $a$ is multiplied with every coefficients of $b$; each thread accumulates $s$ partial sums into an auxiliary array $M$.




*Addition phase*: these partial sums are added together repeatedly to form the polynomial $f$.



# Plain univariate polynomial multiplication

The work, span and parallelism overhead ratios between $s_0 = 1$ (initial program) and an arbitrary $s$ are, respectively, [2]

$$\frac{\mathrm{W}_1}{\mathrm{W}_s} = \frac{n}{n+s-1},$$

$$\frac{\mathrm{S}_1}{\mathrm{S}_s} = \frac{\log_2(m)+1}{s\left(\log_2\left(m/s\right)+2\,s-1\right)},$$

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

- Increasing $s$ leaves work essentially constant, while span increases and parallelism overhead decreases in the same order when $m$ escapes to infinity.
- Hence, should $s$ be large or close to $s_0 = 1$?

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

# Plain univariate polynomial multiplication

Applying our version of the Graham-Brent theorem, the ratio $R$ of the estimated running times on $\Theta(\frac{(n+s-1)\,m}{\ell\,s^2})$ SMs is

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

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.

# Plain univariate polynomial multiplication

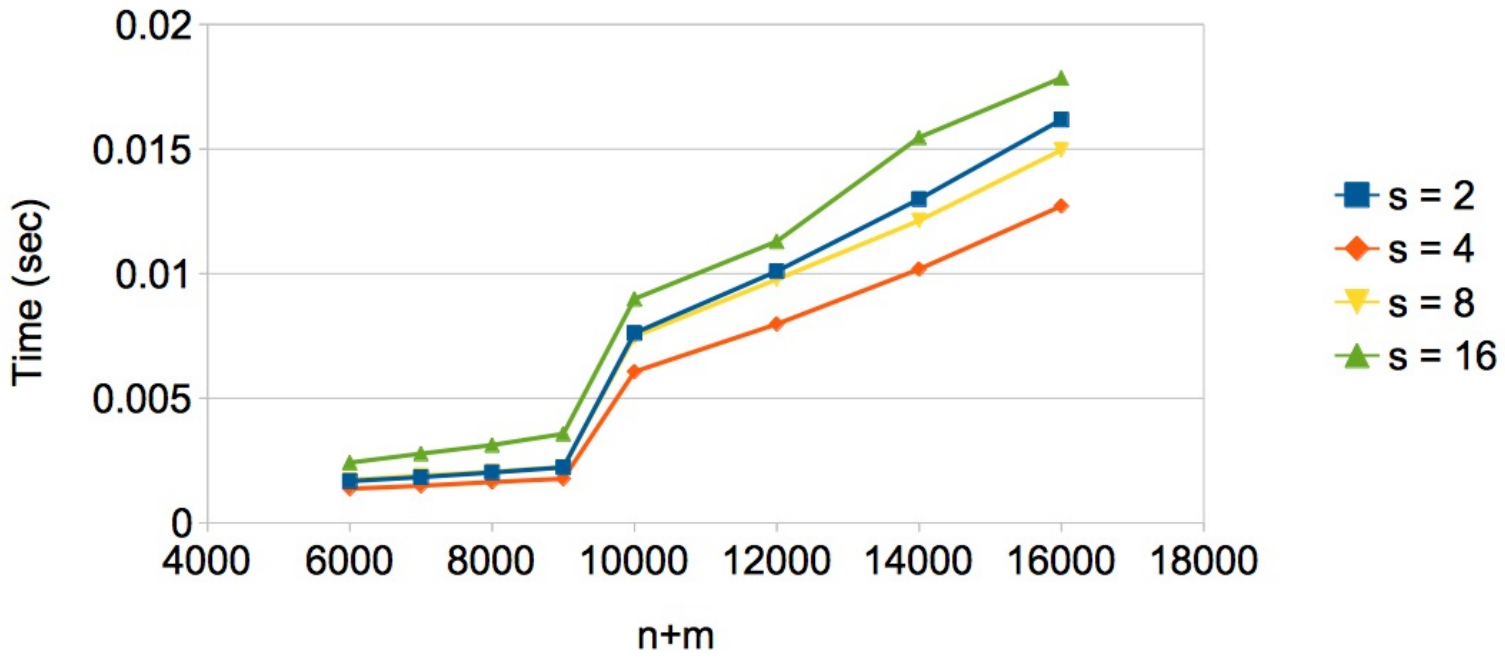
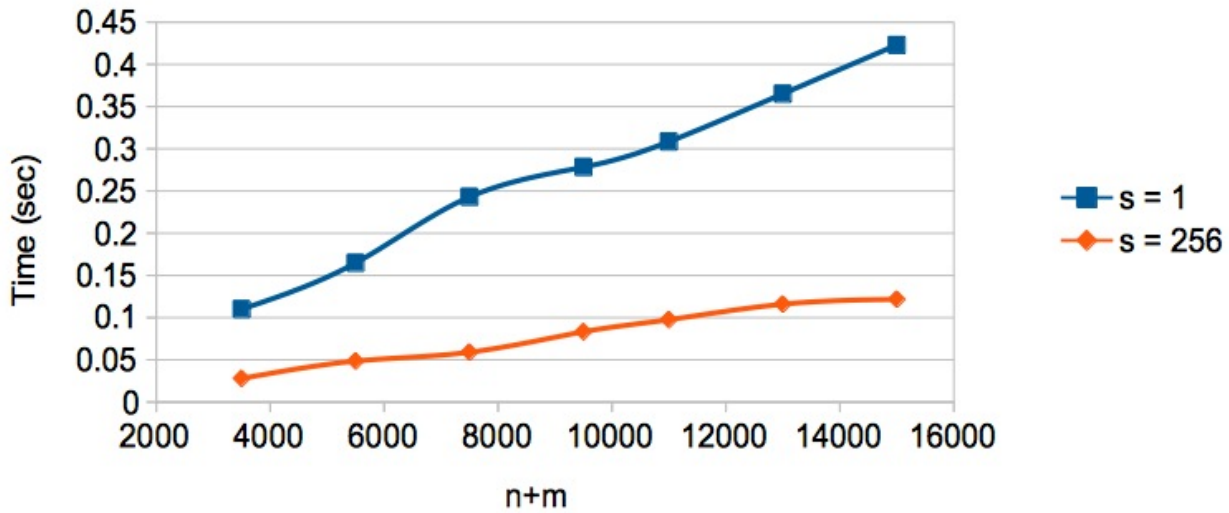



Figure: Running time of the plain polynomial multiplication algorithm with polynomials $a$ ($\deg(a) = n - 1$) and $b$ ($\deg(b) = m - 1$) and the parameter $s$ on GeForce GTX 670.

# The Euclidean algorithm

Let $s > 0$. We proceed by repeatedly calling a subroutine which

- takes as input a pair $(a, b)$ of polynomials and
- returns another pair $(a', b')$ of polynomials such that $\gcd(a, b) = \gcd(a', b')$ and, either $b' = 0$ or we have $\deg(a') + \deg(b') \leq \deg(a) + \deg(b) - s$.
- When $s = \Theta(\ell)$ (the number of threads per thread-block), the work is increased by a constant factor and the parallelism overhead will reduce by a factor in $\Theta(s)$.
- Further, the estimated running time ratio $T_1/T_s$ on $\Theta(\frac{m}{\ell})$ SMs is greater than 1 if and only if $s > 1$.



# Fast Fourier Transform

Let $f$ be a vector with coefficients in a field (either a prime field like $\mathbb{Z}/p\mathbb{Z}$ or $\mathbb{C}$) and size $n$, which is a power of 2. Let $\omega$ be a $n$-th primitive root of unity.

The $n$-point *Discrete Fourier Transform* (DFT) at $\omega$ is the linear map defined by $x \longmapsto \mathrm{DFT}_n x$ with

$$\mathrm{DFT}_n = \left[\omega^{ij}\right]_{0 \leq i,j < n}.$$

We are interested in comparing popular algorithms for computing DFTs on many-core architectures:

- Cooley & Tukey FFT algorithm,
- Stockham FFT algorithm.

# Fast Fourier Transforms: Cooley & Tukey vs Stockham

The work, span and parallelism overhead ratios between Cooley & Tukey's and Stockham's FFT algorithms are, respectively,

$$\frac{\mathrm{W}_{ct}}{\mathrm{W}_{sh}} \sim \frac{4\,n\,(47\,\log_2(n)\,\ell + 34\,\log_2(n)\,\ell\,\log_2(\ell))}{172\,n\,\log_2(n)\,\ell + n + 48\,\ell^2},$$

$$\frac{\mathrm{S}_{ct}}{\mathrm{S}_{sh}} \sim \frac{34\,\log_2(n)\,\log_2(\ell) + 47\,\log_2(n)}{43\,\log_2(n) + 16\,\log_2(\ell)},$$

$$\frac{\mathrm{O}_{ct}}{\mathrm{O}_{sh}} = \frac{8\,n\,(4\,\log_2(n) + \ell\,\log_2(\ell) - \log_2(\ell) - 15)}{20\,n\,\log_2(n) + 5\,n - 4\,\ell},$$

where $\ell$ is the number of threads per thread-block.

- 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.

# Fast Fourier Transforms: Cooley & Tukey vs Stockham

The ratio $R = T_{ct}/T_{sh}$ of the estimated running times (using our Graham-Brent theorem) on $\Theta(\frac{n}{\ell})$ SMs is [3]:

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

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

| $n$ | 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 |

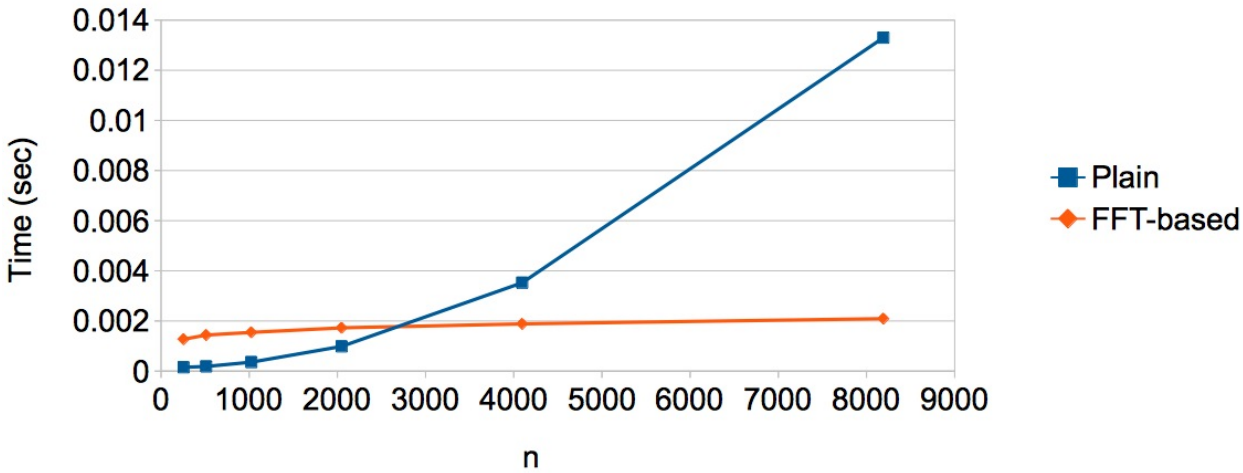
Table: Running time (secs) with input size $n$ on GeForce GTX 670.

[3] $\ell$ is the number of threads per thread-block.

# Univariate polynomial multiplication: Plain vs FFT-based

Polynomial multiplication can be done either via the long (= plain) scheme or via FFT computations.
Let $n$ be the largest size of an input polynomial and $\ell$ be the number of threads per thread-block.

- The theoretical analysis of our model indicates that the plain multiplication performs more work and parallelism overhead.
- However, on $O(\frac{n^2}{\ell})$ SMs, the ratio $T_{plain}/T_{fft}$ of the estimated running times is essentially constant.
- On the other hand, the running time ratio $T_{plain}/T_{fft}$ on $\Theta(\frac{n}{\ell})$ SMs suggests FFT-based multiplication outperforms plain multiplication for $n$ large enough.



# Plan

1. Models of computation
2. A many-core machine (MCM) model
3. Experimental validation
4. Concluding remarks

# Concluding remarks

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.

Intensive experimental validation was conducted with the CUMODP library, which is integrated in the MAPLE computer algebra system.

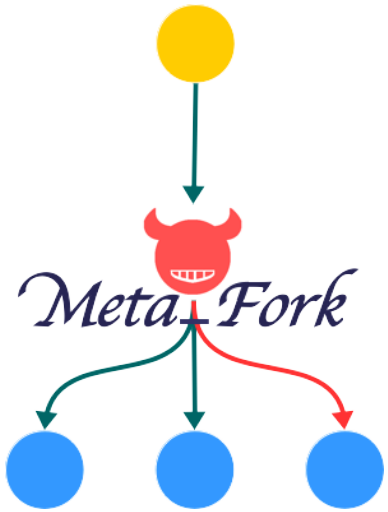

The CUMODP library (`http://www.cumodp.org/`)

# Future works

(1) We plan to integrate our model in the `MetaFork` compilation framework for automatic translation between multithreaded languages.

(2) Within `MetaFork`, we plan to use our model to optimize CUDA-like code automatically generated from `CilkPlus` or `OpenMP` code enhanced with device constructs.

(3) We also plan to extend our model to support hybrid architectures like CPU-GPU.




The Metafork framework (`http://www.metafork.org/`)