

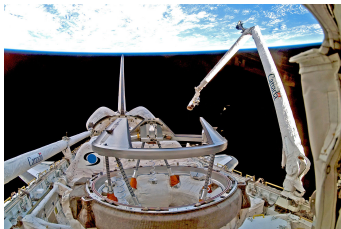
Optimizing Computer Programs: A Killer App for Scientific Computing?

Marc Moreno Maza

University of Western Ontario, Canada

Chongqing Institute of Green and Intelligent Technology
Chinese Academy of Sciences
July 9, 2015

Canada in 4 pictures



High-performance computing and symbolic computation

Research themes

- **Symbolic computation**: computing exact solutions of algebraic problems on computers with applications to mathematical sciences and engineering.
- **High-performance computing**: making best use of modern computer architectures, in particular hardware accelerators (multi-core processors, graphics processing units).

Research projects

- The *RegularChains* library: solving systems of algebraic equations and integrated into the computer algebra system MAPLE.
- The *Basic Polynomial Algebra Subroutines* (BPAS) and *CUDA Modular Polynomial* (CUMODP) libraries: hardware accelerator support for symbolic computation.
- The *Meta_Fork* compilation framework: a programming environment for hardware accelerators, supported by [IBM TORONTO LABS](#).

Current students and alumni

Current students

Parisa Alvandi, Ning Xie, Xiaohui Chen, Li Zhang, Haowei Chen, Yiming Guan, Davood Mohajerani, Steven Thornton and Robert Moir.

Alumni

Moshin Ali (ANU, Australia) Jinlong Cai (Microsoft), Changbo Chen (CIGIT), Sardar Anisula Haque (CiTi) Zunaid Haque (IBM) François Lemaire (U. Lille 1, France) Xin Li (U. Carlos III, Spain) Wei Pan (Intel) Paul Vrbik (U. Newcastle, Australia) Yuzhen Xie (COT) ...

Solving polynomial systems symbolically

$R := \text{PolynomialRing}([x, y, z]); F := [5x^2 + 2xz^2 + 5y^6 + 15y^4 + 5z^2 - 15y^5 - 5y^3];$
polynomial_ring

$[5x^2 + 2xz^2 + 5y^6 + 15y^4 + 5z^2 - 15y^5 - 5y^3]$ (1)

$\text{RealTriangularize}(F, R, \text{output} = \text{record});$

$\begin{cases} 5x^2 + 2z^2x + 5y^6 + 15y^4 - 5y^3 - 15y^5 + 5z^2 = 0 \\ 25y^6 - 75y^5 + 75y^4 - z^4 - 25y^3 + 25z^2 < 0 \end{cases}$ (2)

$\begin{cases} 5x + z^2 = 0 \\ 25y^6 - 75y^5 + 75y^4 - 25y^3 - z^4 + 25z^2 = 0 \\ 64z^4 - 1600z^2 + 25 > 0 \\ z \neq 0 \\ z - 5 \neq 0 \\ z + 5 \neq 0 \end{cases}$

$\begin{cases} x = 0 \\ y - 1 = 0 \\ z = 0 \end{cases}, \begin{cases} x = 0 \\ y = 0 \\ z = 0 \end{cases}, \begin{cases} x + 5 = 0 \\ y - 1 = 0 \\ z - 5 = 0 \end{cases}$

$\begin{cases} x + 5 = 0 \\ y = 0 \\ z - 5 = 0 \end{cases}, \begin{cases} x + 5 = 0 \\ y - 1 = 0 \\ z + 5 = 0 \end{cases}, \begin{cases} x + 5 = 0 \\ y = 0 \\ z + 5 = 0 \end{cases}, \begin{cases} 5x + z^2 = 0 \\ 2y - 1 = 0 \\ 64z^4 - 1600z^2 + 25 = 0 \end{cases}$

Figure: The *RegularChains* solver designed in our UWO lab can compute the real solutions of any polynomial system exactly.

Our polynomial system solver is at the core of MAPLE

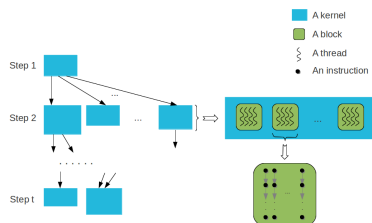
The screenshot shows a web browser window with the URL `http://m/applications/search.aspx?term=Polynomial+system+solving&rcid=&sa.x=0&sa.y=0&sa=Search`. The page title is "Editor's Choice". The main content area is divided into two plots and a text box. The left plot is a 3D surface plot with axes labeled $x(t)$ and $y(t)$, showing a complex surface with a central peak and two side lobes. The right plot is a 2D line plot with axes labeled x and y , showing a family of curves that appear to be solutions to a system of equations. The text box on the right contains the following information:

Polynomial System Solving in Maple 16
Author: [Maplesoft](#)
Maple Document

At the bottom of the page, there is a section titled "Other Editor's Choice" with the instruction "Click thumbnail to display details". This section contains three thumbnails: a 3D bar chart, a 2D plot similar to the one in the main content, and a screenshot of the Maple software interface.

Figure: Maplesoft, the company developing MAPLE, demonstrates the *RegularChains* solver designed in our UWO lab, in order to advertise MAPLE.

High-performance computing: models of computation



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

$$T_{\mathcal{P}} \leq (N(\mathcal{P})/\mathbb{K} + L(\mathcal{P})) C(\mathcal{P}),$$

where $C(\mathcal{P})$ is the maximum running time of local operations by a thread among all the thread-blocks, $N(\mathcal{P})$ is the number of thread-blocks and $L(\mathcal{P})$ is the span of \mathcal{P} .

Our UWO lab develops mathematical models to make efficient use of hardware acceleration technology, such as GPUs and multi-core processors.

High-performance computing: parallel program translation

```
int main(){
  int sum_a=0, sum_b=0;
  int a[ 5 ] = {0,1,2,3,4};
  int b[ 5 ] = {0,1,2,3,4};
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      {
        for(int i=0; i<5; i++)
          sum_a += a[ i ];
      }
      #pragma omp section
      {
        for(int i=0; i<5; i++)
          sum_b += b[ i ];
      } } }
}
```

```
int main()
{
  int sum_a=0, sum_b=0;
  int a[ 5 ] = {0,1,2,3,4};
  int b[ 5 ] = {0,1,2,3,4};

  meta_fork shared(sum_a){
    for(int i=0; i<5; i++)
      sum_a += a[ i ];
  }

  meta_fork shared(sum_b){
    for(int i=0; i<5; i++)
      sum_b += b[ i ];
  }

  meta_join;
}
```

```
void fork_func0(int* sum_a,int* a)
{
  for(int i=0; i<5; i++)
    (*sum_a) += a[ i ];
}

void fork_func1(int* sum_b,int* b)
{
  for(int i=0; i<5; i++)
    (*sum_b) += b[ i ];
}

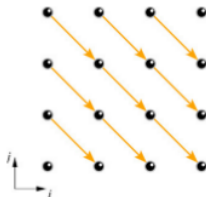
int main()
{
  int sum_a=0, sum_b=0;
  int a[ 5 ] = {0,1,2,3,4};
  int b[ 5 ] = {0,1,2,3,4};
  cilk_spawn fork_func0(&sum_a,a);
  cilk_spawn fork_func1(&sum_b,b);
  cilk_sync;
}
```

Our lab develops a compilation platform for translating parallel programs from one language to another; above we translate from OpenMP to CilkPlus through MetaFork.

High-performance computing: automatic parallelization

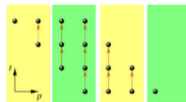
Serial dense univariate polynomial multiplication

```
for(i=0; i<=n; i++){  
  for(j=0; j<=n; j++){  
    c[i+j] += a[i] * b[j];  
  }  
}
```



GPU-like multi-threaded dense univariate polynomial multiplication

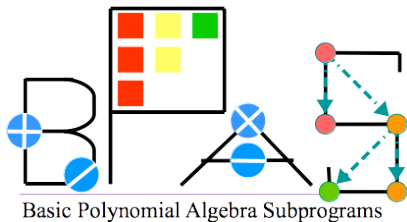
```
meta_for (b=0; b<= 2 n / B; b++) {  
  for (u=0; u<=min(B-1, 2*n - B * b); u++) {  
    p = b * B + u;  
    for (t=max(0,n-p); t<=min(n,2*n-p) ;t++)  
      c[p] = c[p] + a[t+p-n] * b[n-t];  
  }  
}
```



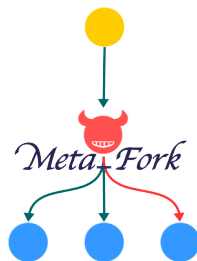
<i>p:</i>	0	1	2	3	4	5	6
<i>r:</i>	0	0	1	1	0	0	1
<i>ac:</i>	0	1	0	1	0	1	0
<i>b:</i>	0	0	1	1	2	2	3
<i>x:</i>	0	0	0	0	1	1	1

We use symbolic computation to automatically translate serial programs to GPU-like programs.

Research projects with publicly available software



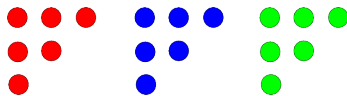
www.bpaslib.org



www.metafork.org

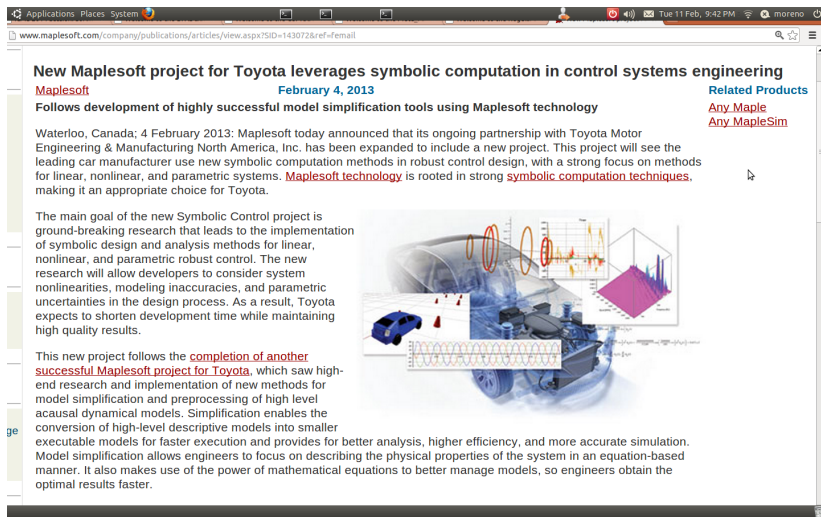
CUMODP $\in \mathbb{F}_p[X_1, \dots, X_s]$
DA \otimes ular polynomial

www.cumodp.org



www.regularchains.org

Application to mathematical sciences and engineering



Applications Places System

www.maplesoft.com/company/publications/articles/view.aspx?SID=143072&ref=femail

New Maplesoft project for Toyota leverages symbolic computation in control systems engineering

Maplesoft February 4, 2013 [Related Products](#)
[Any Maple](#)
[Any MapleSim](#)

Follows development of highly successful model simplification tools using Maplesoft technology

Waterloo, Canada; 4 February 2013: Maplesoft today announced that its ongoing partnership with Toyota Motor Engineering & Manufacturing North America, Inc. has been expanded to include a new project. This project will see the leading car manufacturer use new symbolic computation methods in robust control design, with a strong focus on methods for linear, nonlinear, and parametric systems. [Maplesoft technology](#) is rooted in strong [symbolic computation techniques](#), making it an appropriate choice for Toyota.

The main goal of the new Symbolic Control project is ground-breaking research that leads to the implementation of symbolic design and analysis methods for linear, nonlinear, and parametric robust control. The new research will allow developers to consider system nonlinearities, modeling inaccuracies, and parametric uncertainties in the design process. As a result, Toyota expects to shorten development time while maintaining high quality results.

This new project follows the [completion of another successful Maplesoft project for Toyota](#), which saw high-end research and implementation of new methods for model simplification and preprocessing of high level acausal dynamical models. Simplification enables the conversion of high-level descriptive models into smaller executable models for faster execution and provides for better analysis, higher efficiency, and more accurate simulation. Model simplification allows engineers to focus on describing the physical properties of the system in an equation-based manner. It also makes use of the power of mathematical equations to better manage models, so engineers obtain the optimal results faster.

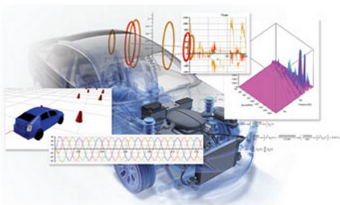


Figure: Toyota engineers use our software to design control systems

Plan

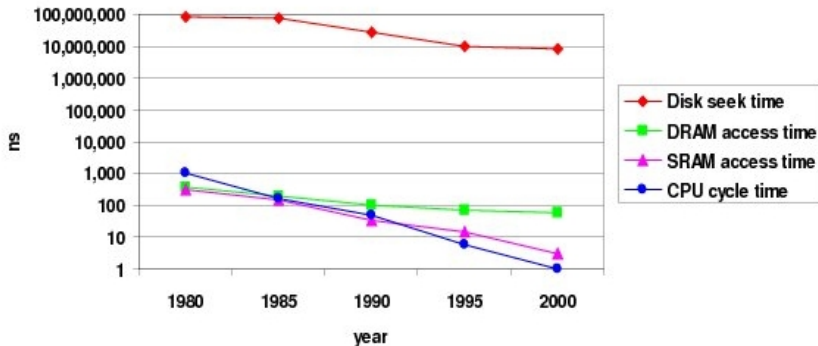
- 1 Optimizing Computer Programs
- 2 GPGPUs and CUDA
- 3 Performance Measures of CUDA Kernels
- 4 Generating Parametric CUDA Kernels
- 5 Experimentation
- 6 Conclusion

Plan

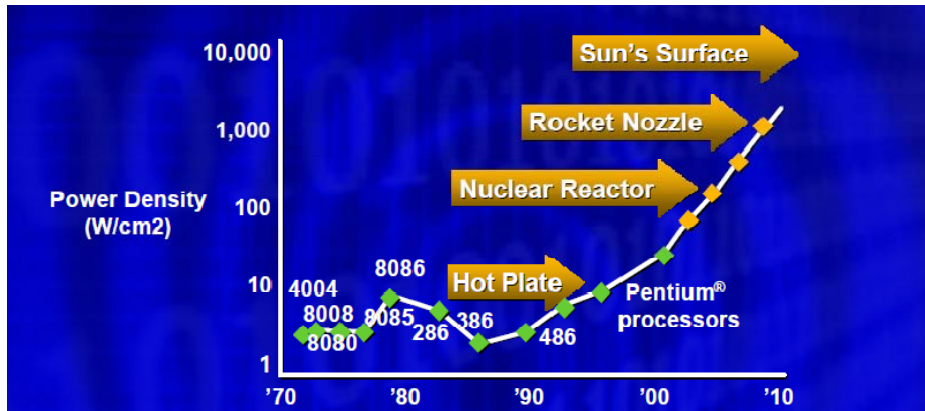
- 1 Optimizing Computer Programs
- 2 GPGPUs and CUDA
- 3 Performance Measures of CUDA Kernels
- 4 Generating Parametric CUDA Kernels
- 5 Experimentation
- 6 Conclusion

The CPU-Memory Gap

The increasing gap between DRAM, disk, and CPU speeds.



Once upon a time, everything was slow in a computer.



The second space race ...

An inefficient program

```
A = (double *)malloc(sizeof(double)*x*y);  
B = (double *)malloc(sizeof(double)*x*z);  
C = (double *)malloc(sizeof(double)*y*z);
```

.....

```
for (i = 0; i < x; i++)  
    for (j = 0; j < y; j++)  
        for (k = 0; k < z; k++)  
            A[i][j] += B[i][k] + C[k][j];
```


A better program

```
A = (double *)malloc(sizeof(double)*x*y);
B = (double *)malloc(sizeof(double)*x*z);
C = (double *)malloc(sizeof(double)*y*z);
Cx = (double *)malloc(sizeof(double)*y*z);
```

.....

```
for(j =0; j < y; j++)
    for(k=0; k < z; k++)
        Cx[j][k] = C[k][j]
for (i = 0; i < x; i++)
    for (j = 0; j < y; j++)
        for (k = 0; k < z; k++)
            A[i][j] += B[i][k] * Cx[j][k];
```

An inefficient program

```
A = (double *)malloc(sizeof(double)*x*y);  
B = (double *)malloc(sizeof(double)*x*z);  
C = (double *)malloc(sizeof(double)*y*z);  
Cx = (double *)malloc(sizeof(double)*y*z);
```

.....

```
for(j =0; j < y; j++)  
    for(k=0; k < z; k++)  
        Cx[j][k] = C[k][j]  
for (i = 0; i < x; i++)  
    for (j = 0; j < y; j++)  
        for (k = 0; k < z; k++)  
            A[i][j] += B[i][k] * Cx[j][k];
```


A nearly optimal program for parallel dense matrix multiplication

```

void parallel_dandc(int i0, int i1, int j0, int j1, int k0, int k1, int* A, int lda, int* B, int ldb, int* C, int ldc,
{
    int di = i1 - i0;
    int dj = j1 - j0;
    int dk = k1 - k0;
    if (di >= dj && di >= dk && di >= X) {
        int mi = i0 + di / 2;
        cilk_spawn parallel_dandc(i0, mi, j0, j1, k0, k1, A, lda, B, ldb, C, ldc,X);
            parallel_dandc(mi, i1, j0, j1, k0, k1, A, lda, B, ldb, C, ldc,X);
        cilk_sync;
    } else if (dj >= dk && dj >= X) {

        int mj = j0 + dj / 2;
        cilk_spawn parallel_dandc(i0, i1, j0, mj, k0, k1, A, lda, B, ldb, C, ldc,X);
            parallel_dandc(i0, i1, mj, j1, k0, k1, A, lda, B, ldb, C, ldc,X);
        cilk_sync;
    } else if (dk >= X) {

        int mk = k0 + dk / 2;
        parallel_dandc(i0, i1, j0, j1, k0, mk, A, lda, B, ldb, C, ldc,X);
        parallel_dandc(i0, i1, j0, j1, mk, k1, A, lda, B, ldb, C, ldc,X);

    } else {
        mm_loop_serial2(C, k0, k1, A, i0, i1, B, j0, j1, lda) ;
        /* for (int i = i0; i < i1; ++i)
           for (int j = j0; j < j1; ++j)
               for (int k = k0; k < k1; ++k)
                   C[i * ldc + j] += A[i * lda + k] * B[k * ldb + j];*/
    }
}

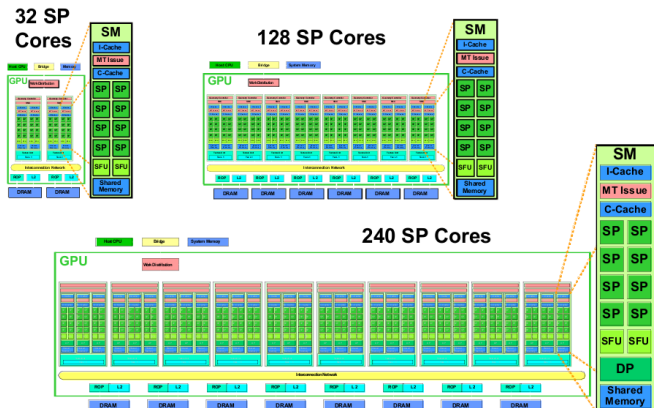
```

Plan

- 1 Optimizing Computer Programs
- 2 GPGPUs and CUDA**
- 3 Performance Measures of CUDA Kernels
- 4 Generating Parametric CUDA Kernels
- 5 Experimentation
- 6 Conclusion

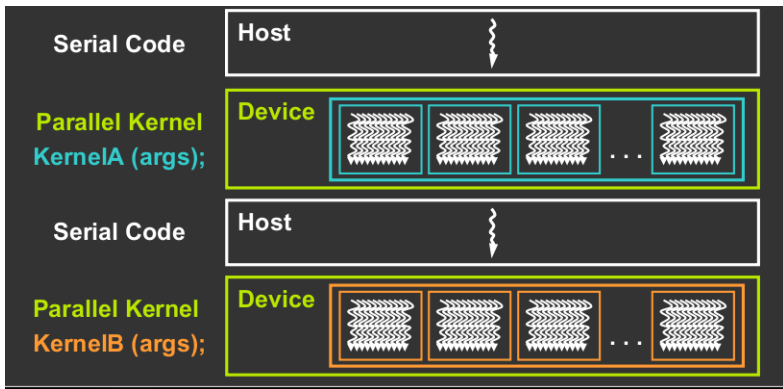
CUDA design goals

- Enable heterogeneous systems (i.e., CPU+GPU)
- Scale to 100's of cores, 1000's of parallel threads
- Use C/C++ with minimal extensions
- Let programmers focus on parallel algorithms (well, that's the intention)



Heterogeneous programming

- A CUDA program is a serial program with parallel kernels, all in C.
- The serial C code executes in a **host** (= CPU) thread
- The parallel kernel C code executes in many **device** threads across multiple GPU processing elements, called **streaming processors** (SP).



Vector addition on GPU (1/4)

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Vector addition on GPU (2/4)

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

```
int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Vector addition on GPU (3/4)

```
// allocate and initialize host (CPU) memory
float *h_A = ..., *h_B = ...; *h_C = ... (empty)

// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),
cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float),
cudaMemcpyHostToDevice) );

// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

Vector addition on GPU (4/4)

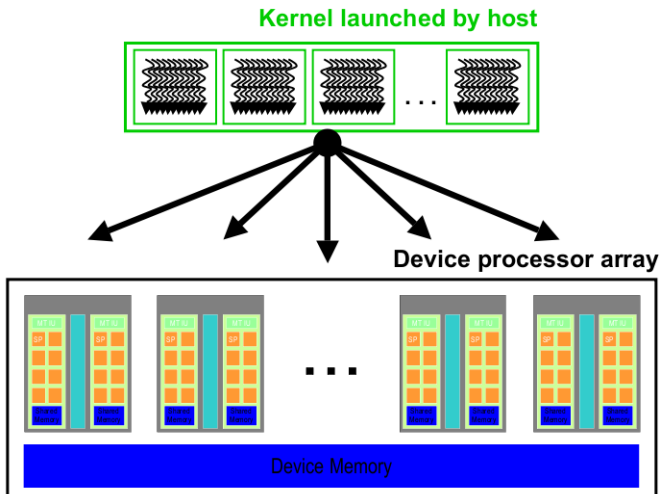
```
// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);

// copy result back to host memory
cudaMemcpy( h_C, d_C, N * sizeof(float),
            cudaMemcpyDeviceToHost) );

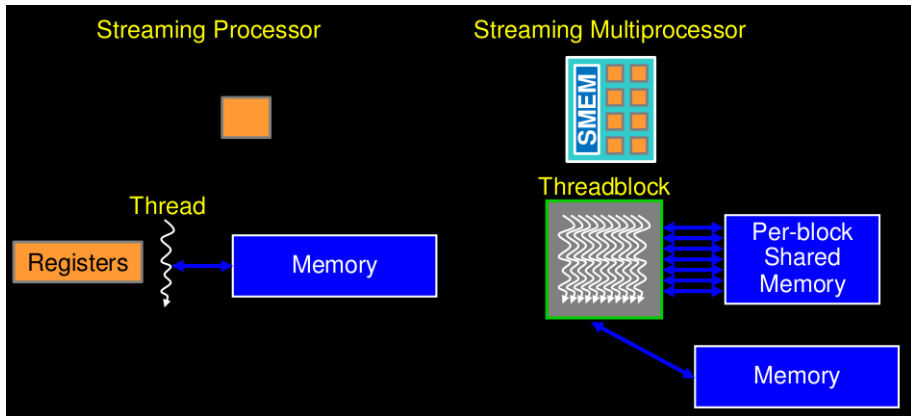
// do something with the result...

// free device (GPU) memory
cudaFree (d_A);
cudaFree (d_B);
cudaFree (d_C);
```

Blocks run on multiprocessors

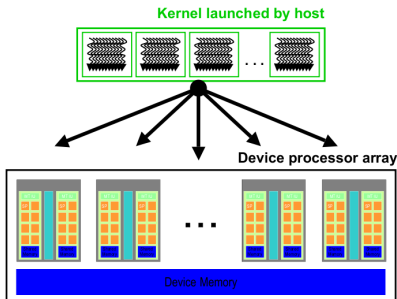


Streaming processors and multiprocessors



Blocks run on multiprocessors: four principles

Hardware allocates resources to blocks and schedules threads.



- ① Expose as much parallelism as possible
- ② Optimize memory usage for maximum bandwidth
- ③ Maximize occupancy to hide latency
- ④ Optimize instruction usage for maximum throughput

Plan

- 1 Optimizing Computer Programs
- 2 GPGPUs and CUDA
- 3 Performance Measures of CUDA Kernels**
- 4 Generating Parametric CUDA Kernels
- 5 Experimentation
- 6 Conclusion

Characteristics of the abstract many-core machines

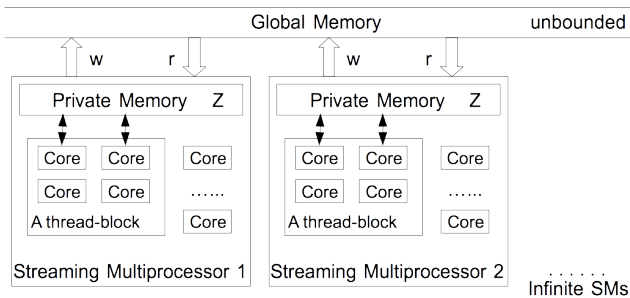


Figure: A many-core machine

- Global memory has high latency and low throughput while private memories have low latency and high throughput.

Characteristics of the abstract many-core machines

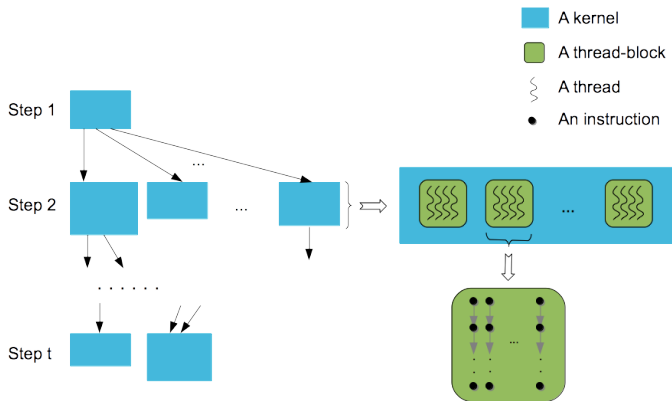


Figure: Sketch of a many-core machine program

Parameters and complexity measures in the MCM model

Machine parameters

- Z : Private memory size of any SM,
- U : Data transfer time.

Program parameters

- ℓ : number of threads per thread-block,
- number of data words read/written per thread,
- ...

Complexity measures

- The **work** accounts for the total amount of local operations (ALU and private memory accesses)
- The **span** accounts for the maximum number of local operations along a path in the DAG representing the program
- The **parallelism overhead** for the total amount of data transfer.

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 ω be a n -th primitive root of unity.

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

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

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,

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

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

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

where ℓ 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 ¹:

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

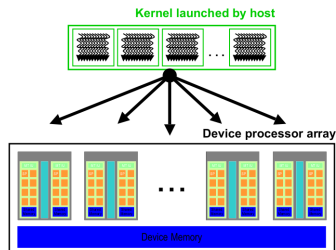
when n escapes to infinity. This latter ratio is greater than 1 if and only if $\ell > 1$.

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.

¹ ℓ is the number of threads per thread-block.

A popular performance counter: occupancy



- The **occupancy** of an SM is $A_{\text{warp}}/M_{\text{warp}}$, where A_{warp} and M_{warp} are respectively the number of active warps and maximum number of running warps per SM.
- warps require resources (registers, shared memory, thread slots) to run
- A_{warp} is bounded over by $M_{\text{block}} \ell$, where M_{block} is the maximum number of active blocks.
- Hence a small value for ℓ may limit occupancy,
- but larger ℓ will reduce the amount of registers and shared memory available per thread; this will limit data reuse within a thread-block.

The need for parametric CUDA kernels

Overall, both theoretical models and empirical performance counters suggest:

- ① Generating kernel code, where ℓ and other parameters are an input arguments.
- ② Once the machine parameters (like S_{warp} , M_{warp} , M_{block} , Z are known, optimize at run-time the values of those program parameters (like ℓ).

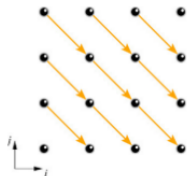
Plan

- 1 Optimizing Computer Programs
- 2 GPGPUs and CUDA
- 3 Performance Measures of CUDA Kernels
- 4 Generating Parametric CUDA Kernels**
- 5 Experimentation
- 6 Conclusion

Automatic parallelization: plain multiplication

Serial dense univariate polynomial multiplication

```
for(i=0; i<=n; i++){
  c[i] = 0; c[i+n] = 0;
  for(j=0; j<=n; j++)
    c[i+j] += a[i] * b[j];
}
```

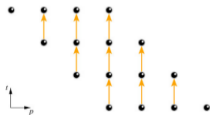


Dependence analysis suggests to set $t(i, j) = n - j$ and $p(i, j) = i + j$.

Synchronous parallel dense univariate polynomial multiplication

```
for (p=0, p<=2*n, p++) c[p]=0;

for (t=0, t=n, t++)
  meta_for (p=n-t; p<=2*n -t; p++)
    c[p] = c[p] + a[t+p-n] * b[n-t];
}
```

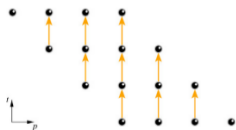


Generating parametric code & use of tiling techniques (1/2)

```

meta_for (p=0; p<=2*n; p++){
  c[p]=0;
  for (t=max(0,n-p); t<= min(n,2*n-p); t++){
    C[p] = C[p] + A[t+p-n] * B[n-t];
  }
}

```



Improving the parallelization

- Make the maximum number of thread-blocks a parameter N .
- We group the virtual processors (or threads) into 1D blocks, each of size B . Each thread is known by its block number b and a local coordinate u in its block.
- Blocks represent good units of work which have good locality property. The total number of blocks may exceed N so blocks are processed in a cyclic manner; the cycle index is s .
- We have: $0 \leq r \leq N - 1$, $b = sN + r$, $0 \leq u < B$, $p = bB + u$.

Generating parametric code: using tiles (2/2)

Let us generate CUDA-like code. Hence we do not need to worry about scheduling the blocks and we just schedule the threads within each block. Thus we only consider the following relations on the left to which we apply our QE tools (in order to get rid off i, j) leading to the relations on the right

$$\left\{ \begin{array}{l} o < n \\ 0 \leq i \leq n \\ 0 \leq j \leq n \\ t = n - j \\ p = i + j \\ 0 \leq b \\ 0 \leq u < B \\ p = bB + u, \end{array} \right. \quad \left\{ \begin{array}{l} B > 0 \\ n > 0 \\ 0 \leq b \leq 2n/B \\ 0 \leq u < B \\ 0 \leq u \leq 2n - Bb \\ p = bB + u, \end{array} \right. \quad (1)$$

from where we derive the following program:

```
for (p=0; p<=2*n; p++) c[p]=0;
meta_for (b=0; b<= 2 n / B; b++) {
    meta_for (u=0; u<=min(B-1, 2*n - B * b); u++) {
        p = b * B + u;
        for (t=max(0,n-p); t<=min(n,2*n-p) ;t++)
            c[p] = c[p] + a[t+p-n] * b[n-t];
    }
}
```

Generation of parametric parallel programs

Summary

- Given a parallel algorithm (e.g. divide-and-conquer matrix multiplication) expressed in METAFORK with **program parameters** like B ,
- given a type of hardware accelerators, like GPGPUs, characterized by **machine parameters**, like Z, M ,
- we can:
 - ① automatically generate code that depends on the machine and program parameters Z, M, \dots, B , by means of **symbolic computation**,
 - ② specialize the machine parameters for a specific accelerator of the above type,
 - ③ optimize the program parameters by means of **numerical computation**.

Note

The symbolic computation part, which is a special form of **quantifier elimination (QE)** (Changbo Chen & M³, ISSAC 2014 & CASC 2015) is performed by our RegularChains library in MAPLE available at

Plan

- 1 Optimizing Computer Programs
- 2 GPGPUs and CUDA
- 3 Performance Measures of CUDA Kernels
- 4 Generating Parametric CUDA Kernels
- 5 Experimentation**
- 6 Conclusion

A complete example: Jacobi

```
for (int t = 0; t < T; ++t) {  
    for (int i = 1; i < N-1; ++i)  
        b[i] = (a[i-1] + a[i] + a[i+1]) / 3;  
  
    for (int i = 1; i < N-1; ++i)  
        a[i] = b[i];  
}
```

Original C code.

A complete example: Jacobi

```

int ub_v = (N - 2) / B;

meta_schedule {
  for (int t = 0; t < T; ++t) {
    meta_for (int v = 0; v < ub_v; v++) {
      meta_for (int u = 0; u < B; u++) {
        int p = v * B + u + 1;
        int y = p - 1;
        int z = p + 1;
        b[p] = (a[y] + a[p] + a[z]) / 3;
      }
    }
    meta_for (int v = 0; v < ub_v; v++) {
      meta_for (int u = 0; u < B; u++) {
        int w = v * B + u + 1;
        [w] = b[w];
      }
    }
  }
}

```

METAFORK code obtained via quantifier elimination.

A complete example: Jacobi

```

#include "jacobi_kernel.hu"
__global__ void kernel0(int *a, int *b, int N,
                       int T, int ub_v, int B, int c0)
{
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    int private_p;
    int private_y;
    int private_z;
    extern __shared__ int shared_a[];

#define floord(n,d) (((n)<0) ? -((-n)+(d)-1)/(d)) : (n)/(d)
    for (int c1 = b0; c1 < ub_v; c1 += 32768) {

        if (!t0) {
            shared_a[(B)] = a[(c1 + 1) * (B)];
            shared_a[(B) + 1] = a[(c1 + 1) * (B) + 1];
        }
        if (N >= t0 + (B) * c1 + 1)
            shared_a[t0] = a[t0 + (B) * c1];
        __syncthreads();
        for (int c2 = t0; c2 < B; c2 += 512) {
            private_p = (((c1) * (B)) + (c2)) + 1);
            private_y = (private_p - 1);
            private_z = (private_p + 1);
            b[private_p] = (((shared_a[private_y - (B) * c1] +
                               shared_a[private_p - (B) * c1]) +
                               shared_a[private_z - (B) * c1]) / 3);
        }
        __syncthreads();
    }
}

```

CUDA kernel corresponding to the first loop nest.

Preliminary implementation

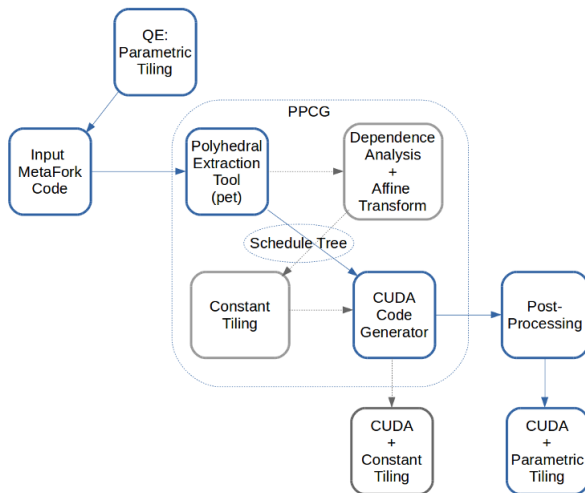


Figure: Components of METAfork-to-CUDA generator of parametric code.

Reversing an array

Speedup (kernel)	Input size				
Block size	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}
8	1.695	1.689	1.610	1.616	1.623
16	3.267	3.330	3.120	3.125	3.130
32	6.289	6.513	6.389	6.432	6.276
64	11.385	11.745	11.792	11.811	11.776
128	16.116	17.191	17.976	18.202	18.333
256	14.464	17.108	17.614	19.704	20.264
512	13.956	14.216	16.002	17.171	18.249

Table: Reversing a one-dimensional array by METAFORK.

Speedup (kernel)	Input size				
Block size	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}
32	7.844	8.202	8.225	8.024	8.140

Table: Reversing a one-dimensional array by PPCG.

Matrix transpose

Speedup (kernel)	Input size		
Block size	2^{12}	2^{13}	2^{14}
8 * 8	17.940	19.138	20.896
8 * 16	12.384	14.857	17.571
16 * 4	16.792	20.275	32.335
16 * 8	19.107	21.365	29.859
16 * 16	10.665	13.888	18.729
16 * 32	4.930	6.047	7.713
32 * 4	19.257	22.397	30.549
32 * 8	15.947	17.402	25.209
32 * 16	10.509	12.114	15.011

Table: Matrix transpose by METAFORK.

Speedup (kernel)	Input size		
Block size	2^{12}	2^{13}	2^{14}
16 * 32	50.060	63.102	104.787

Table: Matrix transpose by PPCG.

Matrix addition

Speedup (kernel)	Input size	
	2^{12}	2^{13}
Block size		
4 * 4	14.419	14.870
4 * 8	33.103	32.679
4 * 16	35.432	33.270
4 * 32	20.202	18.445
8 * 8	44.995	44.860
8 * 16	42.257	35.772
16 * 4	43.960	47.065
16 * 8	44.841	53.761
16 * 16	43.847	33.112
16 * 32	15.238	8.927
32 * 4	54.460	49.150
32 * 8	44.505	39.024
32 * 16	30.300	18.571

Table: Matrix addition by METAFORK.

Speedup (kernel)	Input size	
	2^{12}	2^{13}
Block size		
16 * 32	5.651	5.018

Table: Matrix addition by PPCG.

Matrix multiplication

Speedup (kernel)	Input size	
	2^{11}	2^{12}
Block size		
4 * 4	22.180550	25.989502
4 * 8	39.404386	51.298267
4 * 16	73.658984	95.768978
4 * 32	43.123198	54.494699
8 * 4	34.486799	44.828359
8 * 8	111.695654	142.638582
8 * 16	128.915358	166.739415
8 * 32	69.528568	89.975465
16 * 4	50.311660	64.348409
16 * 8	101.053287	130.995876
16 * 16	110.964165	144.353908
16 * 32	69.064474	90.188780

Table: Matrix multiplication by METAFORK.

Speedup (kernel)	Input size	
	2^{11}	2^{12}
Block size		
16 * 32	218.668	284.659

Table: Matrix multiplication by PPCG.

Plan

- 1 Optimizing Computer Programs
- 2 GPGPUs and CUDA
- 3 Performance Measures of CUDA Kernels
- 4 Generating Parametric CUDA Kernels
- 5 Experimentation
- 6 Conclusion

Concluding remarks

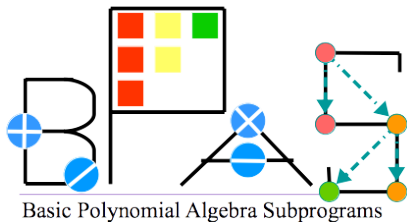
Observations

- Most computer programs that we write are far to make an efficient use of the targeted hardware
- CUDA has brought supercomputing to the desktop computer, but is hard to optimize even to expert programmers.
- High-level models for accelerator programming, like OpenACC, OpenCL and METAFORK are an important research direction.

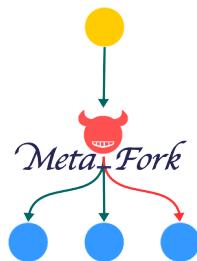
Project

- METAFORK-to-CUDA generates kernels depending on program parameters (like number of threads per block) and machine parameters (like shared memory size) are allowed.
- This is feasible thanks to techniques from [symbolic computation](#).
- Machine parameters and program parameters can be respectively determined and optimized, once the generated code is installed on the target machine.
- The optimization part can be done from [numerical computation](#).

Our project web sites



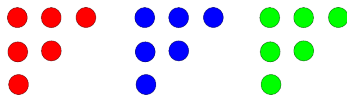
www.bpaslib.org



www.metafork.org

CUMODP $\in \mathbb{F}_p[X_1, \dots, X_s]$
 DA \otimes ular polynomial

www.cumodp.org



www.regularchains.org