Contributions to Automatic Parallelization Probably in Support of Computer Algebra

Marc Moreno Maza

Ontario Research Center for Computer Algebra
Departments of Computer Science and Mathematics
University of Western Ontario, Canada

RTCA 2023, ENS Lyon, France, June 29

Acknowledgements

- Many thanks to the organizers of this workshop for their invitation.
- This talk is based on research projects of my former PhD students: Alexander Brandt (University of Nova Scotia), Sardar Anisul Haque (Otterbein University) Ruijuan Jing (Jiangsu University) Xiaohui Chen (HUAWEI), Davood Mohajerani (Untether AI), Wei Pan (NVIDIA, Delaram Talaashrafi (NVIDIA), Linxiao Wang (HUAWEI), Ning Xie (HUAWEI),
- This talk is also based on collaborations with Maplesoft, MIT/CSAIL, Intel, IBM Canada, Lawrence Livermore National Laboratory with funding support from Maplesoft, IBM Canada, and NSERC of Canada.
- Special thanks go to Alexander Brandt who helped who is leading the development of the Basic Polynomial Algebra Subprograms (BPAS) [1].

Tentative Plan

1. Parallel programming patterns
2. Their BPAS implementation (this part is skipped)
3. Parallelization and automatic parallelization
4. Pipeline pattern detection
5. Automatic GPU offloading
6. Automatic determination of optimal launch parameters for GPU kernels

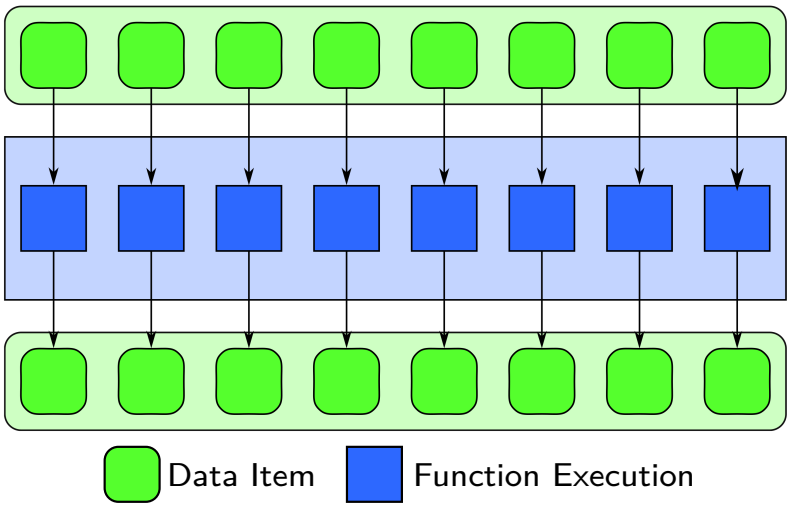
Outline

Parallel map and workpile

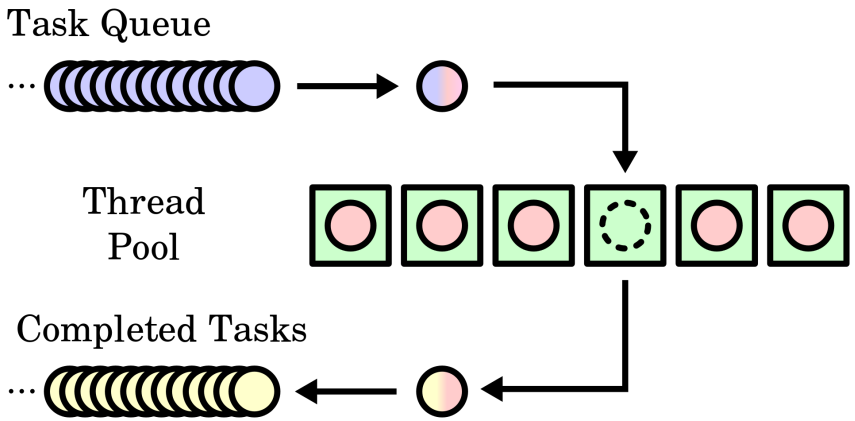
Map is the possibly the best known parallel programming pattern

- ↳ It executes a function on each item in a collection concurrently.
- ↳ With multiple Maps, tasks may execute in *lockstep*.

Map Pattern [16]



Thread Pool (Wikipedia)



Workpile generalizes Map to a *queue of a tasks*, allowing tasks to add more tasks, thus enabling *load-balancing* as tasks start asynchronously

- ↳ one possible implementation of workpile is a **thread pool**

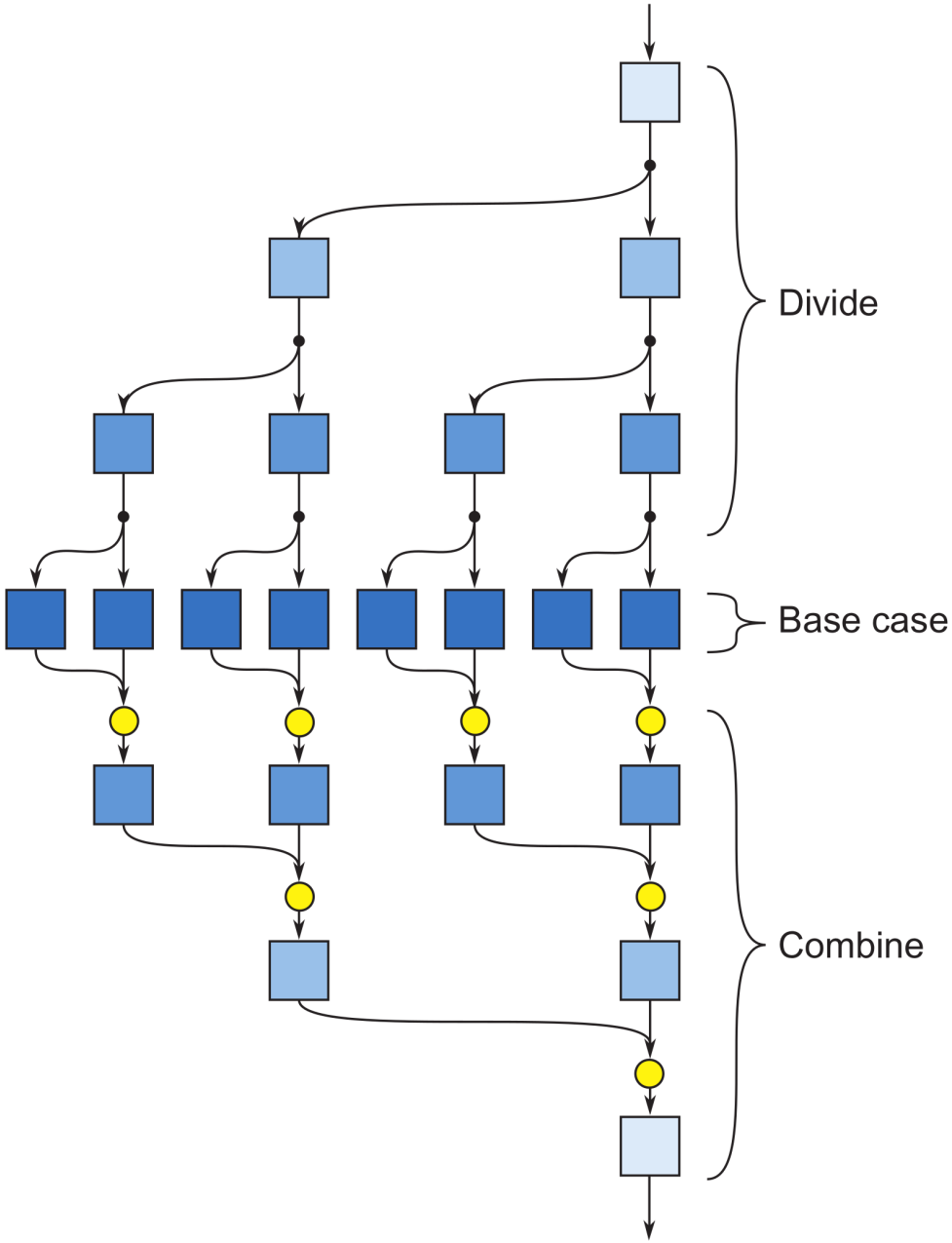
Parallel map and workpile in computer algebra

Opportunities exist and have been studied in

- completion algorithms (Buchberger algorithm, the characteristic set method, etc.)
- dynamic evaluation (computations with regular chains)

Divide-and-conquer and fork-join

- Divide a problem into sub-problems, solving each recursively
- Combine sub-solutions to produce a full solution
- **Fork**: execute multiple recursive calls in parallel (divide)
- **Join**: merge parallel execution back into serial execution (combine)



Divide-and-conquer and fork-join in computer algebra

Opportunities exist and have been intensively studied in

- plain dense linear algebra and plain dense polynomial algebra,
- FFTs,
- asymptotically fast algorithms for matrices, polynomials, series, etc.
- real root isolation, etc.

Generators and pipelines

Generators

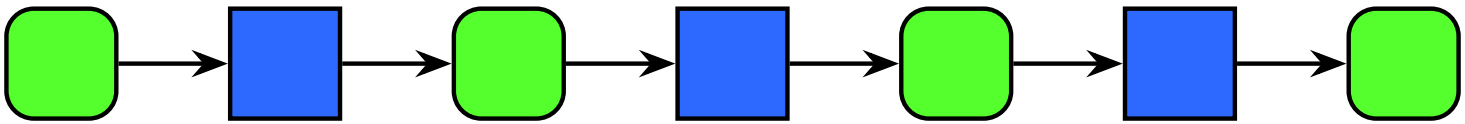
- A generator function (i.e. iterator) yields data items one at a time, allowing the function's control flow to resume on its next execution.

Asynchronous Generators, Producer-Consumer

- *async generators* can concurrently produce items while the generator's caller is consuming items, creating a producer-consumer pair

Pipeline

- By connecting many producer-consumer pairs we create a *pipeline*
- Pipelines need not be linear, they can be *directed acyclic graphs*



Generators and pipelines in computer algebra

Opportunities exist and but have received too little attention:

- linear pipeline: factorization over a finite field with its usual 3 stages (squarefree, distinct degree, equal degree),
- non-linear pipeline: computations of inverses and polynomial GCDs over direct product of fields with (at most) $2n$ stages for towers of n field extensions,
- another non-linear pipeline: iterative methods in numerical linear algebra.

In this talk, we are interested in discovering pipelines from a sequence of for-loop nests accessing/updating arrays.

⏩ skip slide

Outline

Compiler-Level automatic parallelization

- CILK and OPENMP provide automatic parallelization though compiler extensions
- Very easy but flexibility more challenging

```
void mergeSort(int* A, int i,
    int j) {
  //... base case, k
  cilk_spawn mergeSort(A, i, k);
  mergeSort(A, k, j);
  cilk_sync
  merge(A, i, k, j);
}
```

```
void mergeSort(int* A, int i, int
    j) {
  //... base case, k
  #pragma omp parallel
      num_threads(2)
  {
    #pragma omp sections {
      #pragma omp section {
        mergeSort(A, i, k);
      }
      #pragma omp section {
        mergeSort(A, k, j);
      }
    }
  }
  merge(A, i, k, j);
}
```

Fork-Join parallelism with BPAS

- Object-oriented
- Standard C++, no compiler extensions
- Extends the *Thread Support Library* of C++11

```
1  void mergeSort(int* A, int i, int j) {
2      //... base case, k
3      threadID id;
4      ExecutorThreadPool& pool =
5          ExecutorThreadPool::getThreadPool();
6
7      pool.obtainThread(id);
8      pool.executeTask(id, std::bind(mergeSort, A, i, k));
9      mergeSort(A, k, j);
10
11     pool.returnThread(id);
12
13     merge(A, i, k, j);
14 }
```

Outline

Threading primitives

C++11 introduced the *Thread Support Library*

- `std::thread`
  - ↳ C++ class encapsulating a thread (often a `pthread`) and its low-level spawn and join
- `std::mutex`
  - ↳ shared object between threads to indicate *mutual exclusion* to a **critical region**.
  - ↳ `mutex` is *locked* or *owned* by at most one thread at a time.
- `std::lock_guard`, `std::unique_lock`
  - ↳ temporary object wrapping a `mutex` whose object lifetime automatically locks and unlocks the `mutex`.
  - ↳ the constructor **blocks** and only returns once the shared `mutex` is successfully owned by the calling thread.
- `std::condition_variable`
  - ↳ blocks the current thread and temporarily releases a lock
  - ↳ receives notification from another thread to awaken the blocked thread

std::function

Functors, function objects, callable objects

- First-class objects which are callable using normal function syntax
- Are often constructed by passing function names, function pointers
- `std::bind` binds arguments to a function or function object, returning a function object which requires fewer arguments

```
1  void printInteger(int a) {
2      std::cout << a << std::endl;
3  }
4
5  //Function object from function name
6  std::function<void(int)> f_printInt(printInteger);
7  f_printInt(12);
8
9  //Function object binding arguments to function name
10 std::function<void()> f_print42( std::bind(printInteger,42) );
11 f_print42();
```

Parallel overheads

Creating and managing multiple threads of execution can be expensive

- Every thread spawn requires non-insignificant amount of time
- If more threads are active than the hardware supports, **over-subscription** occurs and repeated **context switching** slows down the program
- Thread synchronization, locking `mutexs`, accessing critical regions require special care

Thread pools mitigate the first two, by supplying a fixed number of long-running threads.

Parallel programming patterns are algorithmic designs for efficient thread scheduling and minimizing locking

Outline

Long-Running threads

Threads typically terminate once their assigned function/code block finishes

In order to implement and benefit from a thread pool, we require a mechanism which allows threads to:

1. Remain active until explicitly told to exit (or the entire program exits)
2. Receive new code blocks to execute on demand

`FunctionExecutorThread`s are such long-running threads which receive functions or code blocks and executes them asynchronously.

FunctionExecutorThread usage

```
1  int A[N];
2  int* ret = new int();
3  FunctionExecutorThread t;
4
5  t.sendRequest( [=]() void -> {
6      int s = 0;
7      for (int i = 0; i < N; ++i) {
8          s += A[i];
9      }
10     *ret = s;
11 });
12
13 doSomethingElse();
14
15 //make sure result is available before continuing
16 t.waitForThread();
17
18 std::cout << "sum: " << *ret << std::endl;
```

Object streams

The key to the implementation of `FunctionExectorThread` is the `AyncObjectStream` class. It provides:

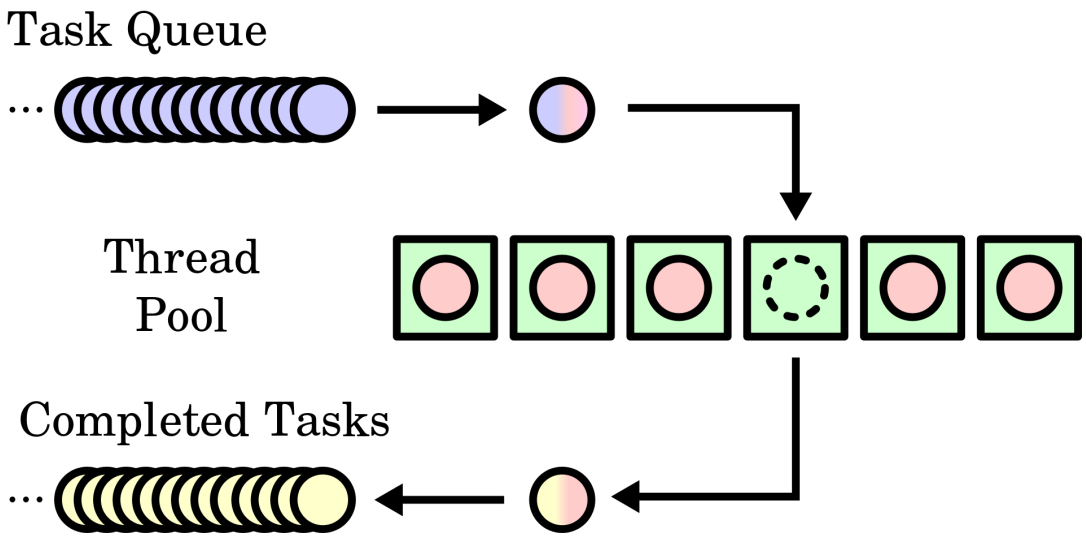
1. a queue for tasks (or any object) and
2. a blocking mechanism to keep the `FunctionExecutorThread` alive and idle when waiting for tasks

- Actually a class template for any kind of object being passed between two threads
- Implements a queue satisfying the **producer-consumer problem**
- A **std::queue** combined with a mutex and condition variable

Thread pools

A **thread pool** manages a collection of long-running threads and a queue of tasks

- spawn all threads once at the beginning of program
- idle threads receive and execute tasks as required
- if all threads busy, tasks are added to queue



ExecutorThreadPool

- A thread pool built using **FunctionExecutorThreads**
- An internal queue of tasks and queue of threads
- When threads are busy, they are temporarily removed from the pool
- When all threads busy, tasks are added to task queue

```
1  class ExecutorThreadPool {
2
3  private:
4      std::deque<FunctionExecutorThread*> threadPool;
5      std::deque<std::function<void()>> taskPool;
6      std::mutex m_mutex;
7      std::condition_variable m_cv; //used in waitForThreads
8
9      void tryPullTask();
10     void putBackThread(FunctionExecutorThread* t);
11
12 public:
13     void addTask(std::function<void()> f);
14     void waitForThreads();
15 }
```

ExecutorThreadPool: flexible usage

- In support of certain **parallel patterns**, clients can (temporarily) obtain ownership of threads from the pool, rather than using `addTask`
- Abstract away actual threads through **thread IDs**
- Once thread obtained, repeat Steps 2–3 as often as necessary

```
1  class ExecutorThreadPool {
2    //Storage for threads removed from pool by obtainThread
3    std::vector<FunctionExecutorThread*> occupiedThreads;
4
5    //Step 1: obtain a thread's ID, removing it from the pool
6    void obtainThread(threadID& id);
7
8    //Step 2: execute a task on a particular thread
9    void executeTask(threadID id, std::function<void()>& f);
10
11   //Step 3 (optional): wait for thread to become idle
12   void waitForThread(threadID id);
13
14   //Step 4: return thread to pool (waits before returning)
15   void returnThread(threadID id);
16 }
```

Outline

Fork-Join with ExecutorThreadPool

```
1  void mergeSort(int* A, int i, int j) {
2      if (j <= i) { return; }
3      int k = i + (j-1)/2;
4      mergeSort(A, i, k);
5      mergeSort(A, k, j);
6      merge(A, i, k, j);
7  }
```

```
1  void mergeSort(int* A, int i, int j) {
2      if (j <= i) { return; }
3      int k = i + (j-1)/2;
4      threadID id;
5      ExecutorThreadPool& pool = getThreadPool();
6
7      pool.obtainThread(id);
8      pool.executeTask(id, std::bind(mergeSort, A, i, k));
9      mergeSort(A, k, j);
10
11     pool.returnThread(id);
12     merge(A, i, k, j);
13 }
```

Workpile with ExecutorThreadPool

```
1  void processInt(std::queue<int> B, int a) {
2      a -= 10;
3      if (a > 0) {
4          getThreadPool().addTask(std::bind(processInt, B, a));
5      } else {
6          B.push(a);
7      }
8  }
9
10 void WorkpileExample(std::queue<int> B, std::queue<int> A) {
11     ExecutorThreadPool& pool = getThreadPool();
12     while (!A.empty()) {
13         pool.addTask( std::bind(processInt, B, A.front()) );
14         A.pop();
15     }
16     pool.waitForAllThreads();
17 }
```

AsyncGenerator and AsyncObjectStream

We want an *object-oriented* approach to create and use generators.

`AsyncObjectStream` already solves the producer-consumer problem.

- It provides a queue which blocks and notifies the consumer as data is produced, implemented using a condition variable
- As a class template, can be used within `AsyncGenerator` to yield any type of object

```
1  template <class Object>
2  class AsyncObjectStream {
3      void addResult(Object&& res); //Producer
4
5      void resultsFinished(); //Producer
6
7      bool getNextObject(Object& res); //Consumer
8
9      void streamEmpty(); //Consumer
10 };
```

AsyncGenerator

`AsyncGenerator` is itself a class template, templated by `Object`, the type of object to generate.

- The `AsyncGenerator` acts as interface between producer and consumer
- The consumer constructs the `AsyncGenerator`, passing the constructor the producer's function and arguments
- The producer's signature should be:

```
1  void producerFunction(..., AsyncGenerator<Object>&);
```

- The `AsyncGenerator` being constructed inserts itself into the producer's list of arguments so that it has reference to the generator object

AsyncGenerator example

```
1  void FibonacciGen(int n, AsyncGenerator<int>& gen) {
2      int Fn_1 = 0;
3      int Fn = 1;
4      for (int i = 0; i < n; ++i) {
5          gen.generateObject(Fn_1); //yield Fn_1 and continue
6          Fn = Fn + Fn_1;
7          Fn_1 = Fn - Fn_1;
8      }
9      gen.setComplete();
10 }
11
12 void Fib() {
13     int n;
14     std::cin >> n;
15     AsyncGenerator<int> gen(FibonacciGen, n);
16
17     int fib;
18     //get one integer at a time until generator is finished
19     while (gen.getNextObject(fib)) {
20         std::cerr << fib << std::endl;
21     }
22 }
```

Cooperative parallelism

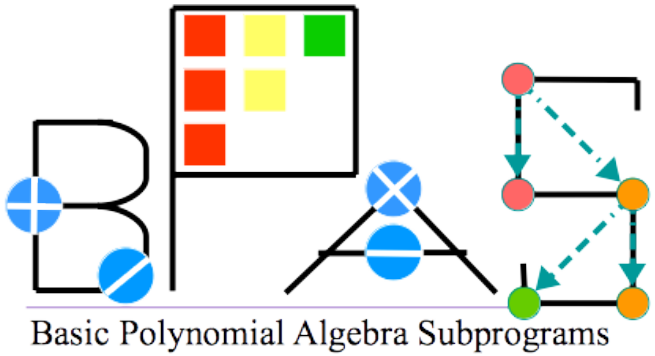
With several simultaneous clients of `ExecutorThreadPool` (workpile, fork-join, generators), some tasks should be given priority.

- Some tasks are more **coarse-grained**, offer more potential speed-up
- Some tasks may expose more parallelism and should be executed first

Often, parallelism coming from Fork-Join or Map is preferred over Producer-Consumer.

- **Goal**: allow Fork-Join and Map to access thread pool threads over Producer-Consumer while still keeping the latter possible when there are idle threads
- **Solution**: **priority tasks**
- `addTask()` vs `addPriorityTask()`
- If all threads busy, `addPriorityTask()` temporarily spawns new thread to start execution immediately

The BPAS library



http://www.bpaslib.org/

A high-performance polynomial algebra library

- Core of library written in C, wrapped in C++ interface for usability and object-oriented programming

Optimized algorithms and data structures, data locality, and parallelism

- Sparse multivariate polynomials [3], dense univariate and bivariate [21]
- Triangular decomposition of polynomial systems [2, 4]

User-friendly, object-oriented interface based on template meta-programming [5]

- A natural encoding of the algebraic hierarchy
- "Dynamic" creation of algebraic types through composition
- Compile-time type safety between algebraic types

Generic support for parallel programming and parallel patterns (this talk)

Outline

Parallelization and automatic parallelization

- Parallelization usually refers to the programmer

Parallelization and automatic parallelization

- Parallelization usually refers to the programmer
  1. applying the fork-join pattern to a divide-and-conquer algorithm, or

Parallelization and automatic parallelization

- Parallelization usually refers to the programmer
  1. applying the fork-join pattern to a divide-and-conquer algorithm, or
  2. turning a **for** into **parallel-for**.

Parallelization and automatic parallelization

- Parallelization usually refers to the programmer
  1. applying the fork-join pattern to a divide-and-conquer algorithm, or
  2. turning a **for** into **parallel-for**.
- Automatic parallelization usually refers to the compiler

Parallelization and automatic parallelization

- Parallelization usually refers to the programmer
  1. applying the fork-join pattern to a divide-and-conquer algorithm, or
  2. turning a **for** into **parallel-for**.
- Automatic parallelization usually refers to the compiler
  1. exposing parallelism (dependence analysis, tiling, loop transformations, etc.)

Parallelization and automatic parallelization

- Parallelization usually refers to the programmer
  1. applying the fork-join pattern to a divide-and-conquer algorithm, or
  2. turning a **for** into **parallel-for**.
- Automatic parallelization usually refers to the compiler
  1. exposing parallelism (dependence analysis, tiling, loop transformations, etc.)
  2. scheduling tasks (load balancing, reducing parallelism overheads, etc.),

Parallelization and automatic parallelization

- Parallelization usually refers to the programmer
  1. applying the fork-join pattern to a divide-and-conquer algorithm, or
  2. turning a **for** into **parallel-for**.
- Automatic parallelization usually refers to the compiler
  1. exposing parallelism (dependence analysis, tiling, loop transformations, etc.)
  2. scheduling tasks (load balancing, reducing parallelism overheads, etc.),
  3. generating parallel code (allocating resources, in particular memory).

Dependence analysis in the polyhedral model

Cholesky's LU decomposition:

```
   for(i = 1; i <= n; i++){
1:      x = a[i][i];
        for(k = 1; k < i; k++)
2:        x = x - a[i][k] * a[i][k];
3:      p[i] = 1.0/sqrt(x);
        for(j = i + 1; j <= n; j++){
4:        x = a[i][j];
          for(k = 1; k < i; k++)
5:          x = x - a[j][k] * a[i][k];
6:        a[j][i] = x * p[i];
        }
    }
```

Dependence analysis in the polyhedral model

Cholesky's LU decomposition:

```
   for(i = 1; i <= n; i++){
1:      x = a[i][i];
        for(k = 1; k < i; k++)
2:        x = x - a[i][k] * a[i][k];
3:      p[i] = 1.0/sqrt(x);
        for(j = i + 1; j <= n; j++){
4:        x = a[i][j];
          for(k = 1; k < i; k++)
5:          x = x - a[j][k] * a[i][k];
6:        a[j][i] = x * p[i];
        }
    }
```

system 1:

$$\begin{cases} 1 \le i \le n \\ i+1 \le j \le n \\ 1 \le k \le i-1 \\ 1 \le i' \le n \\ i'+1 \le j' \le n \\ j = j', k = i' \\ i < i' \end{cases}$$

system 2:

$$\begin{cases} 1 \le i \le n \\ i+1 \le j \le n \\ 1 \le k \le i-1 \\ 1 \le i' \le n \\ i'+1 \le j' \le n \\ j = j', k = i' \\ i = i', j < j' \end{cases}$$

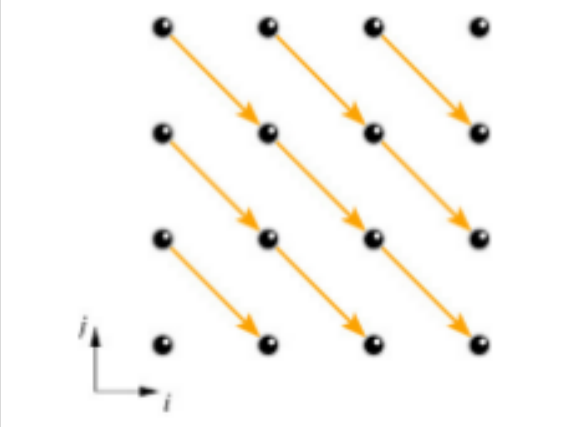
system 3:

$$\begin{cases} 1 \le i \le n \\ i+1 \le j \le n \\ 1 \le k \le i-1 \\ 1 \le i' \le n \\ i'+1 \le j' \le n \\ j = j', k = i' \\ i = i', j = j' \end{cases}$$

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];
}
```

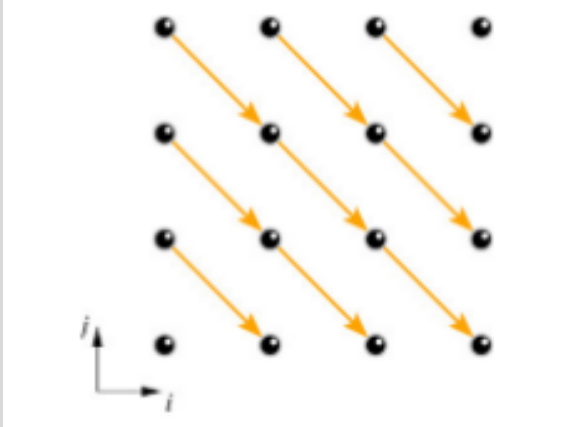


Dependence analysis suggests to set $t(i,j) = n - j$ and $p(i,j) = i + j$.

Automatic parallelization: plain multiplication

Serial dense univariate polynomial multiplication

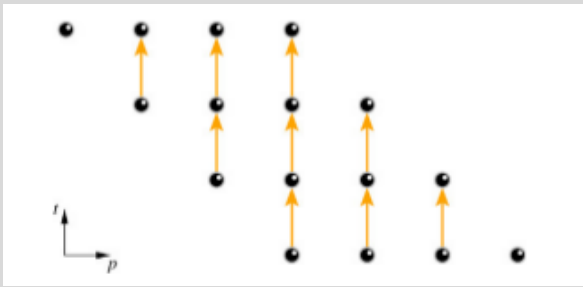
```
for(i=0; i<=n; i++){
  c[i] = 0; c[i+n] = 0;
  for(j=0; j<=n; j++)
    c[i+j] += a[i] * b[j];
}
```



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

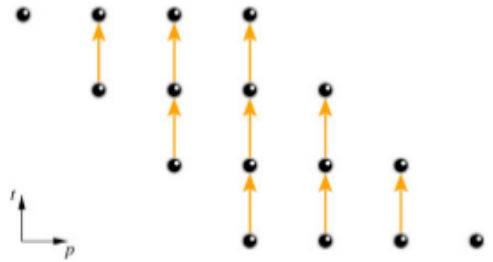
Asynchronous parallel dense univariate polynomial multiplication

```
parallel_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 ] ;
}
```



Generating parametric code & use of tiling techniques

```
parallel_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

- The above generated code is not practical for multicore implementation: the number of processors is in $\Theta(n)$. (Not to mention poor locality!) and the work is unevenly distributed among the workers.
- 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.
- This yields the following constraints: $0 \leq u < B,\ p = bB + u$.

Generating parametric code: using tiles

We apply `Regular-Chains:-QuantifierElimination` on the left system (in order to get rid off i, j) leading to the relations on the right:

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

From where we derive the following program:

```
for (p=0; p<=2*n; p++) c [ p ] = 0;
parallel_for (b=0; b<= 2 n / B; b++) {
    parallel_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 ];
    }
}
```

Outline

This is a joint work with:

- Delaram Talaashrafi (NVIDIA)
- Johannes Doerfert (Lawrence Livermore National Laboratory)

Published in LLPP-2022:

- Delaram Talaashrafi, Johannes Doerfert, and MMM, “A Pipeline Pattern Detection Technique in Polly," in ICPP Workshops ’22: 51th International Conference on Parallel Processing Workshop, Bordeaux, France, August 2022.

The problem

The polyhedral model

- The polyhedral model is effective for optimizing loop nests using different methods: dependence analysis, loop tiling (blocking), ...
- They all optimize for-loop nests on a **per-loop** basis.

Our work

- is about exploiting **cross-loop** parallelization, through tasking.
- is done by detecting pipeline pattern between iteration blocks of different loop nests.

Polly

Polly is an LLVM-based framework, which applies polyhedral transformations: analysis, transformation, scheduling, code generation.

We use **OpenMP** `task` construct and the `depend` clause to exploit the detected parallelism.

Motivating example

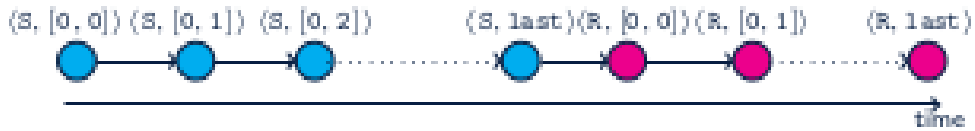
- The Statement `S` of he first loop nest updates the array `A`
- The Statement `T` of he second loop nest updates the array `B` and uses the array `A`
- Before the first loop completes one can start executing the second.

```
for(i=0; i<N-1; i++)
  for(j=0; j<N-1; j++)
  S:  A[i][j]=f(A[i][j], A[i][j+1], A[i+1][j+1]);

for(i=0; i<N/2-1; i++)
  for(j=0; j<N/2-1; j++)
  R:  B[i][j]=g(A[i][2*j], B[i][j+1], B[i+1][j+1],
                B[i][j]);
```

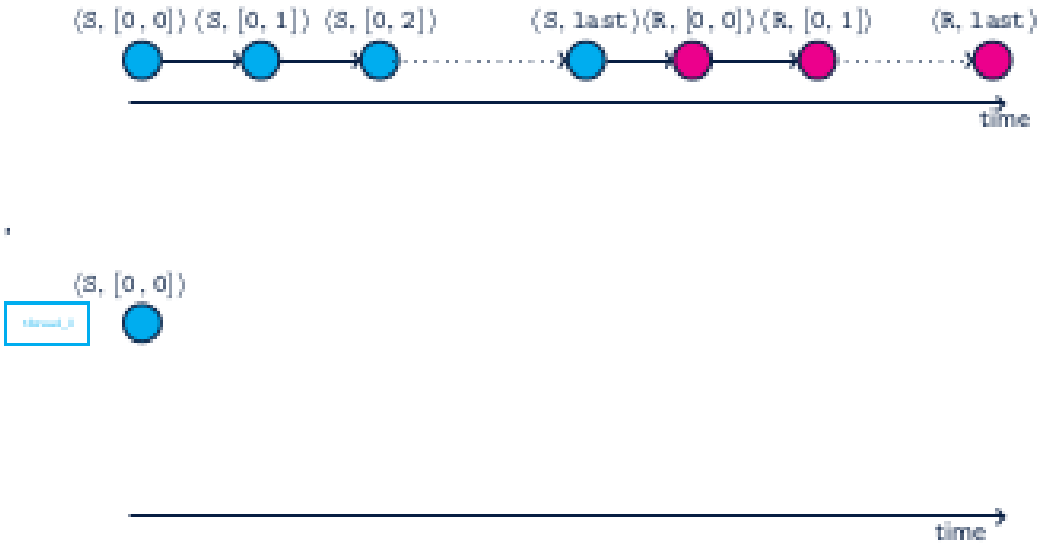
Motivating example

```
1 for(i=0; i<N-1; i++)
2   for(j=0; j<N-1; j++)
3   S:  A[i][j]=f(A[i][j], A[i][j+1], A[i+1][j+1]);
4
5 for(i=0; i<N/2-1; i++)
6   for(j=0; j<N/2-1; j++)
7   R:  B[i][j]=g(A[i][2*j], B[i][j+1], B[i+1][j+1],
8               B[i][j]);
```



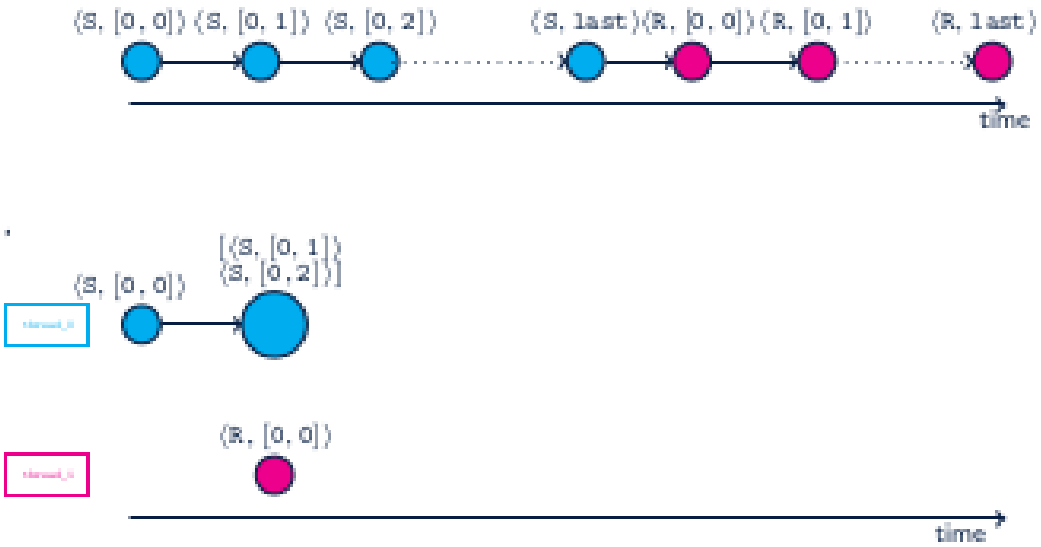
Motivating example

```
1 for(i=0; i<N-1; i++)
2   for(j=0; j<N-1; j++)
3   S:  A[i][j]=f(A[i][j], A[i][j+1], A[i+1][j+1]);
4
5 for(i=0; i<N/2-1; i++)
6   for(j=0; j<N/2-1; j++)
7   R:  B[i][j]=g(A[i][2*j], B[i][j+1], B[i+1][j+1],
8                 B[i][j]);
```



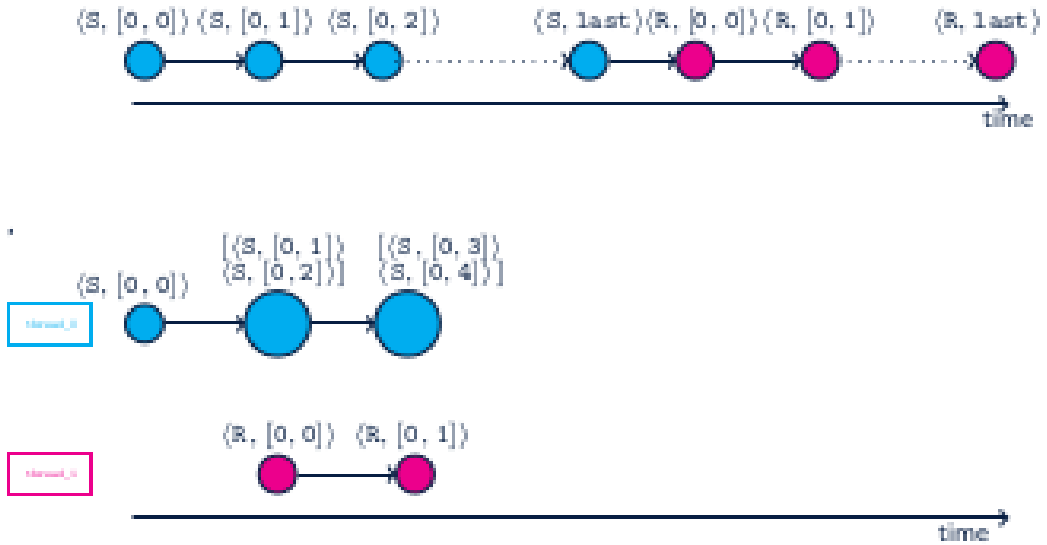
Motivating example

```
1 for(i=0; i<N-1; i++)
2   for(j=0; j<N-1; j++)
3   S:  A[i][j]=f(A[i][j], A[i][j+1], A[i+1][j+1]);
4
5 for(i=0; i<N/2-1; i++)
6   for(j=0; j<N/2-1; j++)
7   R:  B[i][j]=g(A[i][2*j], B[i][j+1], B[i+1][j+1],
8               B[i][j]);
```



Motivating example

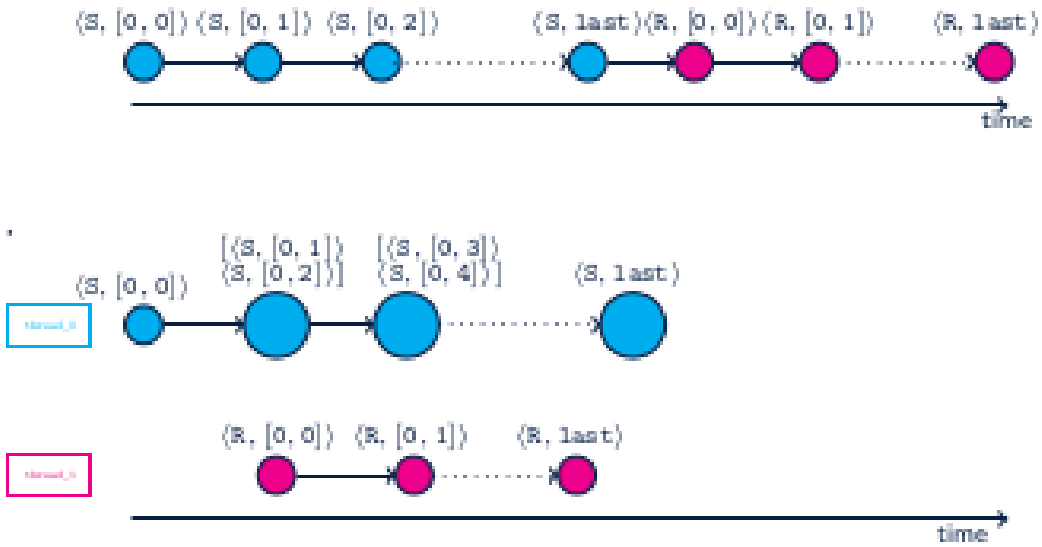
```
1 for(i=0; i<N-1; i++)
2   for(j=0; j<N-1; j++)
3   S:  A[i][j]=f(A[i][j], A[i][j+1], A[i+1][j+1]);
4
5 for(i=0; i<N/2-1; i++)
6   for(j=0; j<N/2-1; j++)
7   R:  B[i][j]=g(A[i][2*j], B[i][j+1], B[i+1][j+1],
8                 B[i][j]);
```



Motivating example

```
for(i=0; i<N-1; i++)
  for(j=0; j<N-1; j++)
  S:  A[i][j]=f(A[i][j], A[i][j+1], A[i+1][j+1]);

for(i=0; i<N/2-1; i++)
  for(j=0; j<N/2-1; j++)
  R:  B[i][j]=g(A[i][2*j], B[i][j+1], B[i+1][j+1],
                B[i][j]);
```



Transformation Algorithm (1/3)

Consider two statements in a program:

- S: with iteration domain $\mathcal{I}$, which writes a memory location in $\mathcal{M}$, thus defining a binary relation $Wr(\mathcal{I} \to \mathcal{M})$
- T: with iteration domain $\mathcal{J}$, which reads a memory location from $\mathcal{M}$, thus defining a binary relation $Rd(\mathcal{J} \to \mathcal{M})$

The **pipeline map** between S and T is $\mathcal{T}_{\mathtt{S},\mathtt{T}}(\mathcal{I} \to \mathcal{J})$, where $(\vec{i}, \vec{j}) \in \mathcal{T}_{\mathtt{S},\mathtt{T}}$ if and only if:

1. after running all iterations of S up to $\vec{i}$, we can safely run all iterations of T up to $\vec{j}$,
2. $\vec{i}$ is the smallest vector and $\vec{j}$ is the largest vector with Property (1).

Transformation Algorithm (2/3)

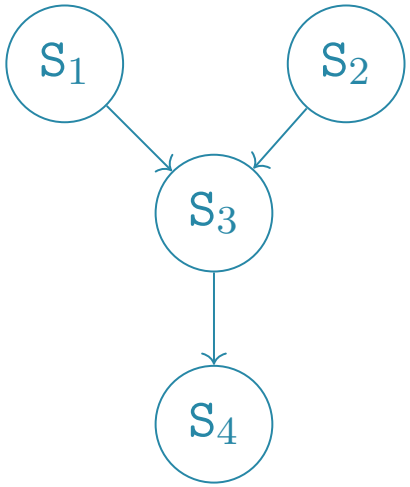
1. We first find the pipeline map between all pairs of dependent statements,
2. then, use them to block the iteration domains and find **pipeline blocking maps**.

The final blocks are such that:

1. each block is an atomic task,
2. we can establish a pipeline relation between all blocks of all statements,
3. maximize the number of blocks of different loops that can execute in parallel.

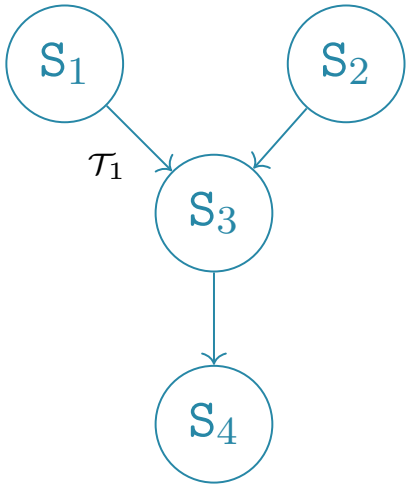
In the last step, we find **dependency relations** between the tasks.

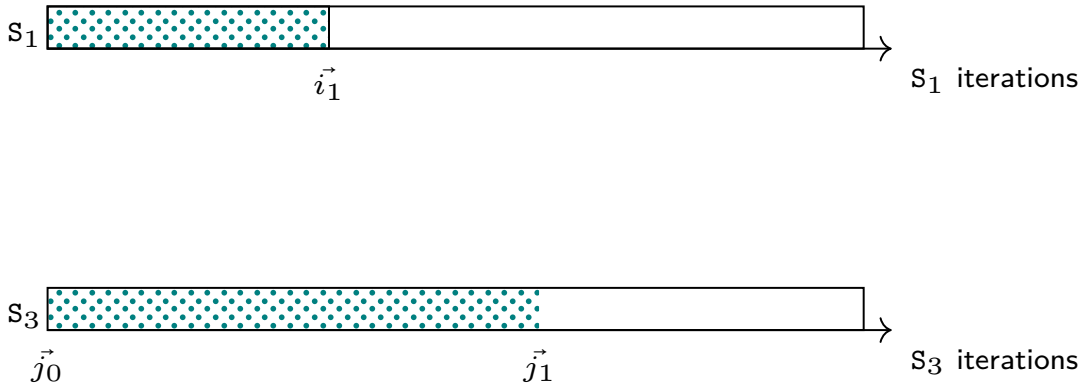
Transformation Algorithm (3/3)



skip slide

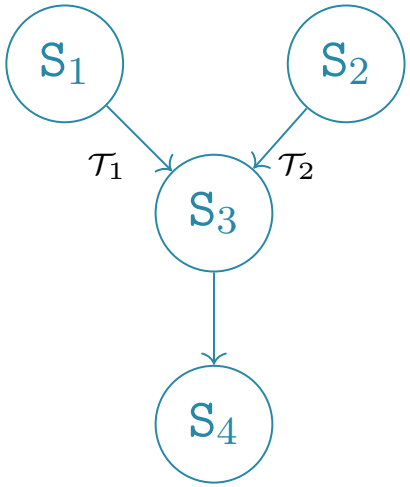
Transformation Algorithm (3/3)

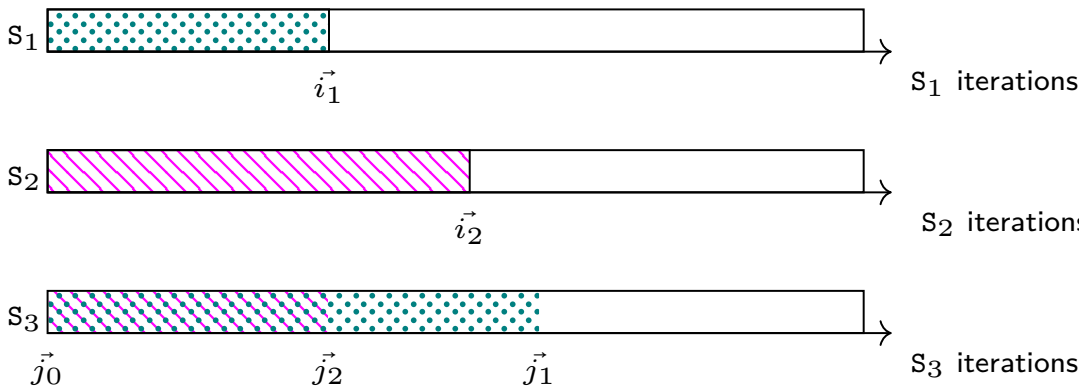




⏩ skip slide

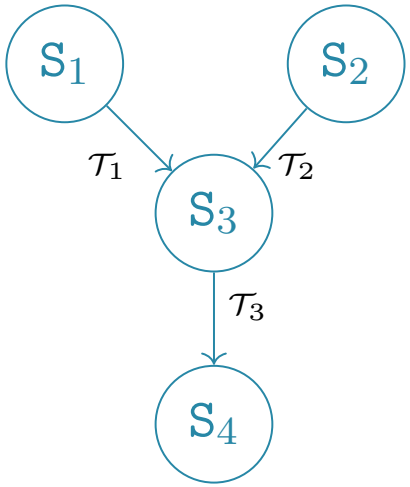
Transformation Algorithm (3/3)

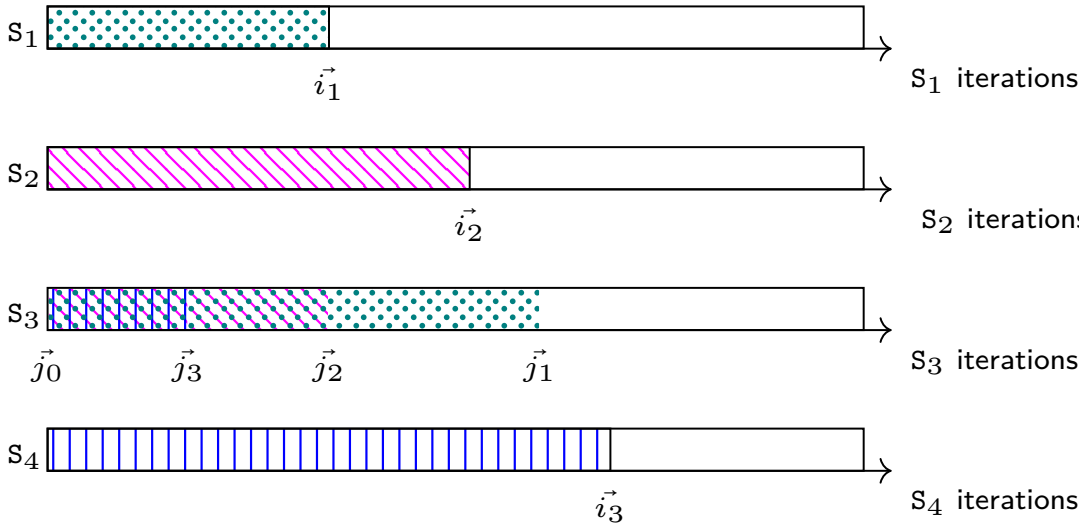




skip slide

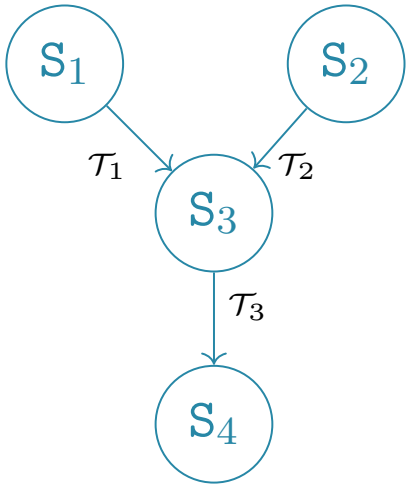
Transformation Algorithm (3/3)

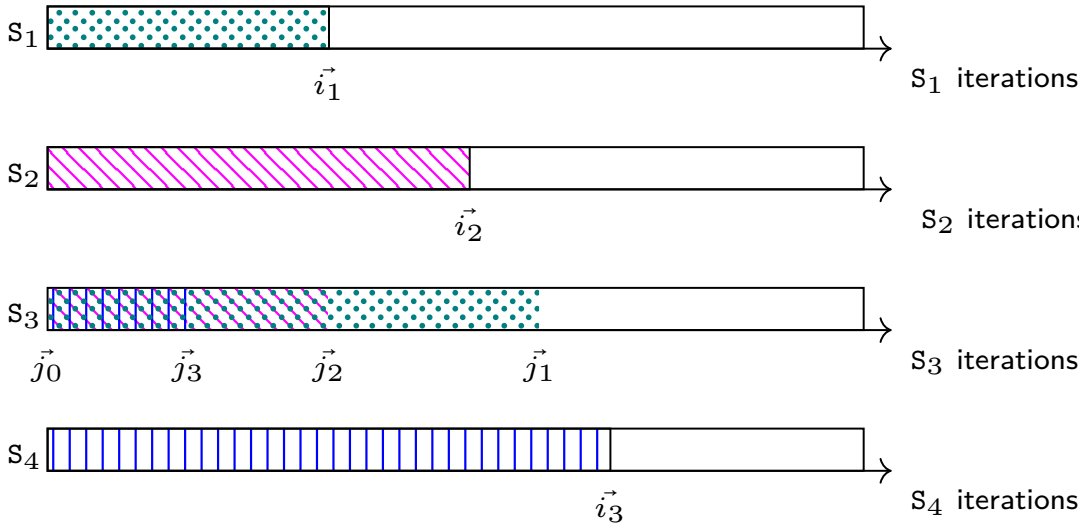




skip slide

Transformation Algorithm (3/3)





Optimal block of S_3: $\langle S_3, \vec{j}_3 \rangle$
Pipeline dependencies: $\langle S_1, \vec{i}_1 \rangle$, $\langle S_2, \vec{i}_2 \rangle$

⏩ skip slide

Implementation (1/2)

Analysis passes of Polly

We **extend** the analysis passes of Polly to compute pipeline information for the iteration domains.

Scheduling

1. Create a schedule tree to iterate **over** blocks,
2. Create a schedule tree to iterate **inside** each block,
3. **Expand** the first tree with the second tree.
4. Create `pw_multi_aff_list` objects from pipeline dependency relations,
5. Add the `pw_multi_aff_list` objects as mark nodes to the schedule tree.

Implementation (2/2)

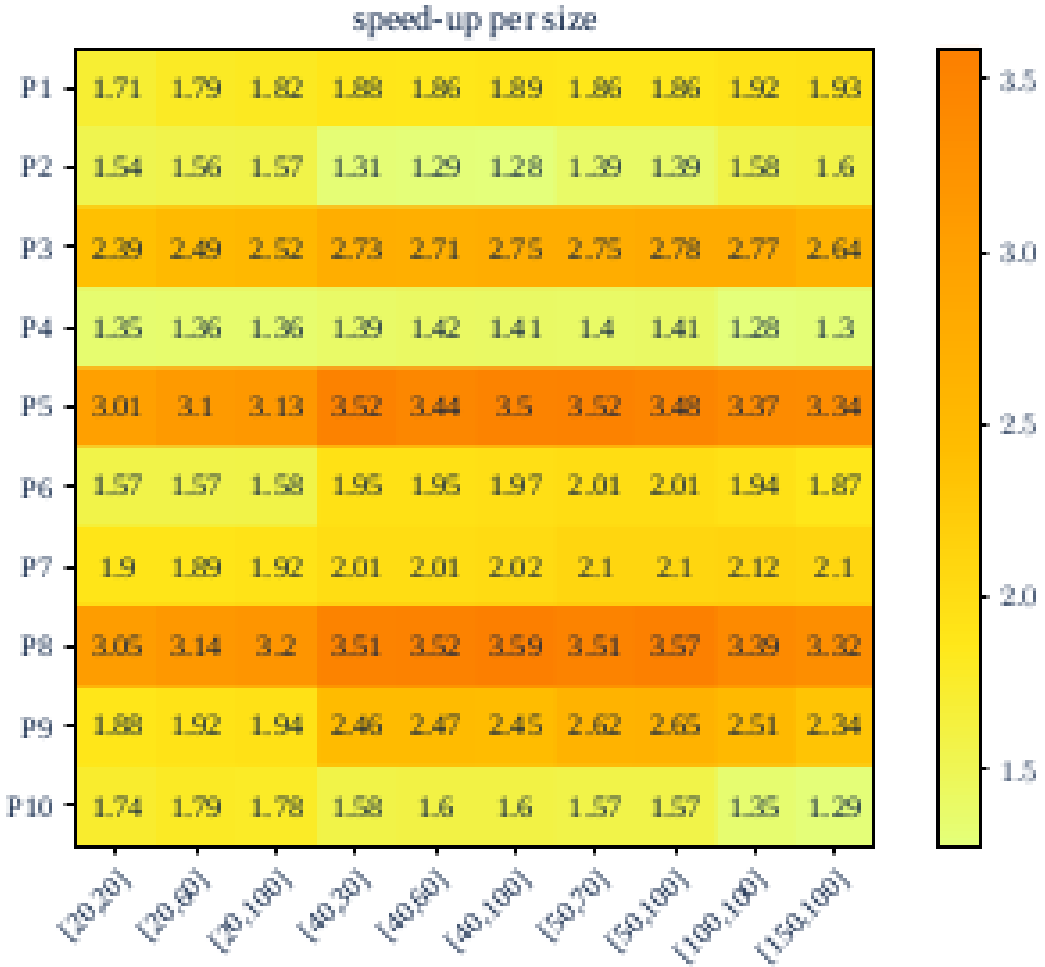
1. We generate AST from the new schedule tree.
2. The mark nodes in the schedule tree **annotates** the AST.

Code generation

1. Outline tasks to function calls,
2. Compute unique integer numbers from `pw_multi_aff_list` objects
   - ↳ this can be used in OpenMP `depend` clauses.
3. Replace the tasks part in the code with call to the `CreateTask` function that:
   - ↳ gets tasks and dependencies, creates OpenMP `tasks` with proper `depend` clauses,
   - ↳ handles the order between tasks created from the same loop nest.

Evaluation

Pipelining speed-up of the tests with different access functions, and different sizes on 4 cores.



Related Works

Pipelined multithreading generation in a polyhedral compiler [22]:

- a limited, polyhedral model based, source-to-source method to solve a similar problem
- uses `nowait` and `order` clauses of OpenMP to exploit pipeline parallelism

Compiling neural networks for a computational memory accelerator [15]

- an algorithm to detect pipeline parallelism between two loop nests
- applicable on the computational memory accelerators considered in that paper

Future works

Generalize **transformation** algorithm:

- works with non-injective write functions,

Generalize our **code generation** phase:

- generate code for loops with arbitrary depth and arbitrary number of tasks per loop.

Develop an algorithm to **choose a good task granularity** when there are multiple choices.

Change the tasking layer from the OpenMP `task` to **other platforms**.

Take advantage of **other parallelization opportunities**, when using the cross-loop tasking.

Outline

- KLARAPTOR: A Tool for Dynamically Finding Optimal Kernel Launch Parameters Targeting CUDA Program.
- This is a joint work with Alexander Brandt, Taabish Jeshani, Davood Mohajerani, Jeeva Paudel (IBM) and Linxiao Wang.
- This is a US Patent (US 10,901,713 B2)
- https://github.com/orcca-uwo/KLARAPTOR

Outline

Vector addition in CUDA

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);
}
```

- Launch parameters greatly impact the performance of a given kernel

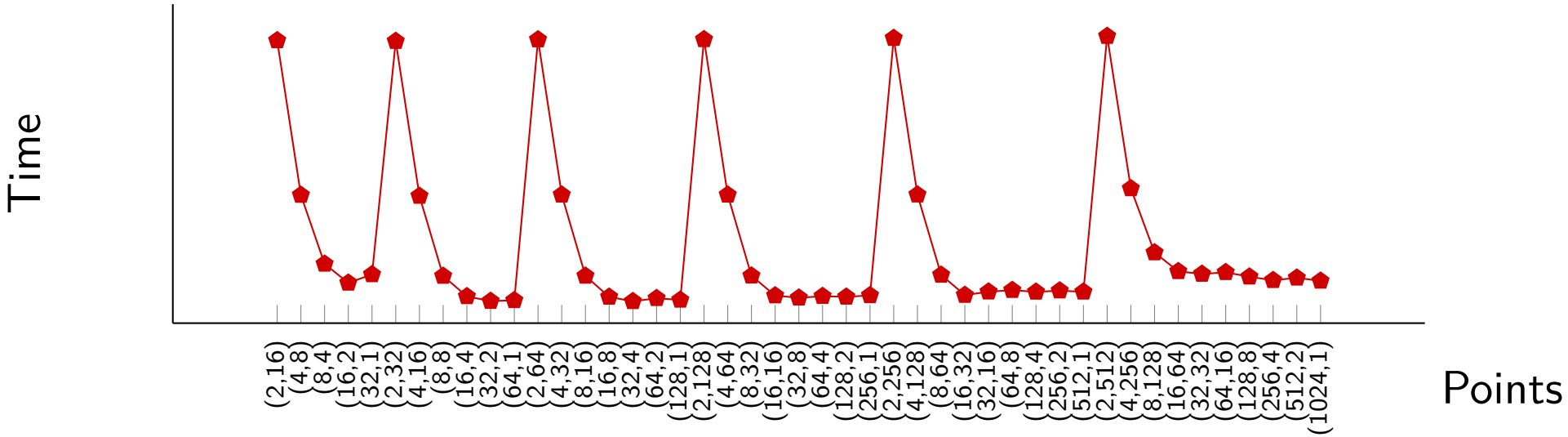


Figure: Example polybench _2DCONV Kernel Convolution2D_kernel for input size 8192

- Launch parameters greatly impact the performance of a given kernel

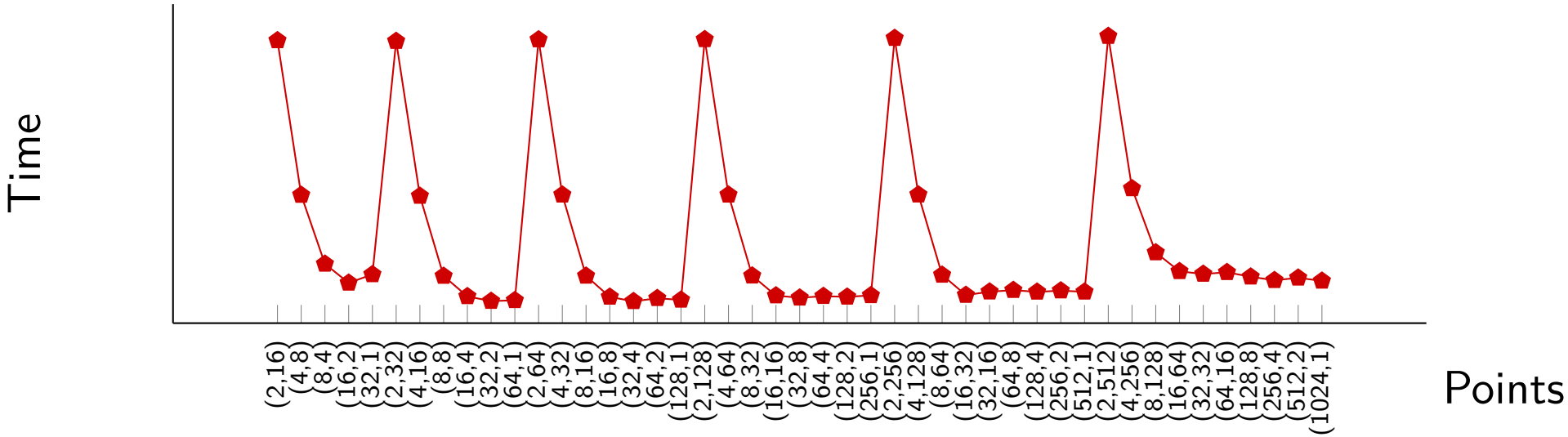


Figure: Example polybench
_2DCONV Kernel Convolution2D_kernel for input size 8192

- They depend on data size and cannot be determined at compile time.

- Launch parameters greatly impact the performance of a given kernel

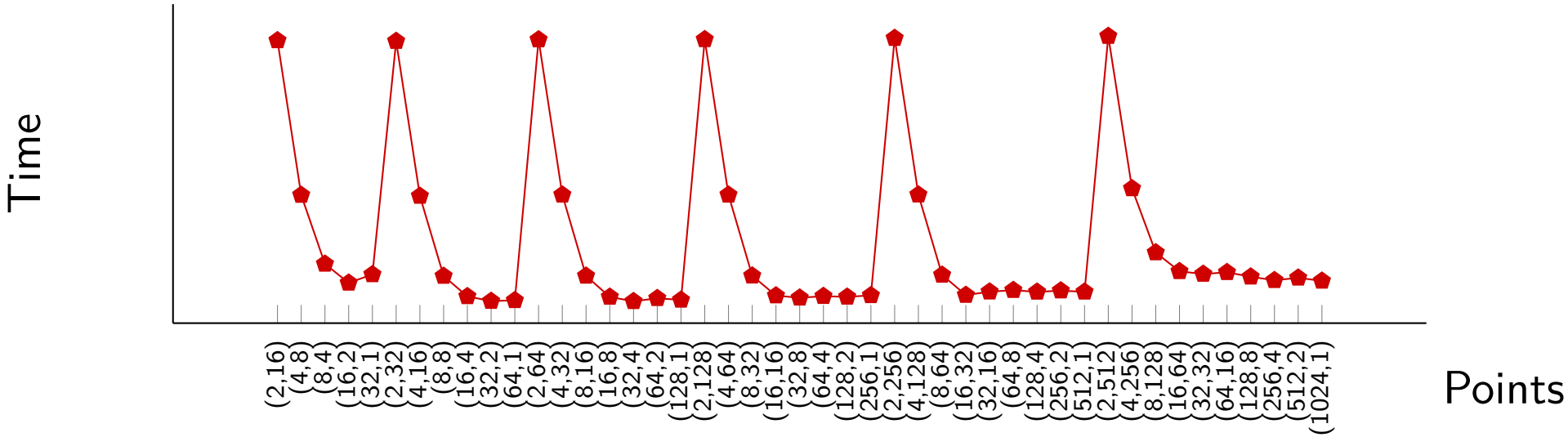


Figure: Example polybench
_2DCONV Kernel Convolution2D_kernel for input size 8192

- They depend on data size and cannot be determined at compile time.
- klaraptor's goal: automatically and dynamically find the values of the launch parameters which optimize the performance of a CUDA kernel.

- Launch parameters greatly impact the performance of a given kernel

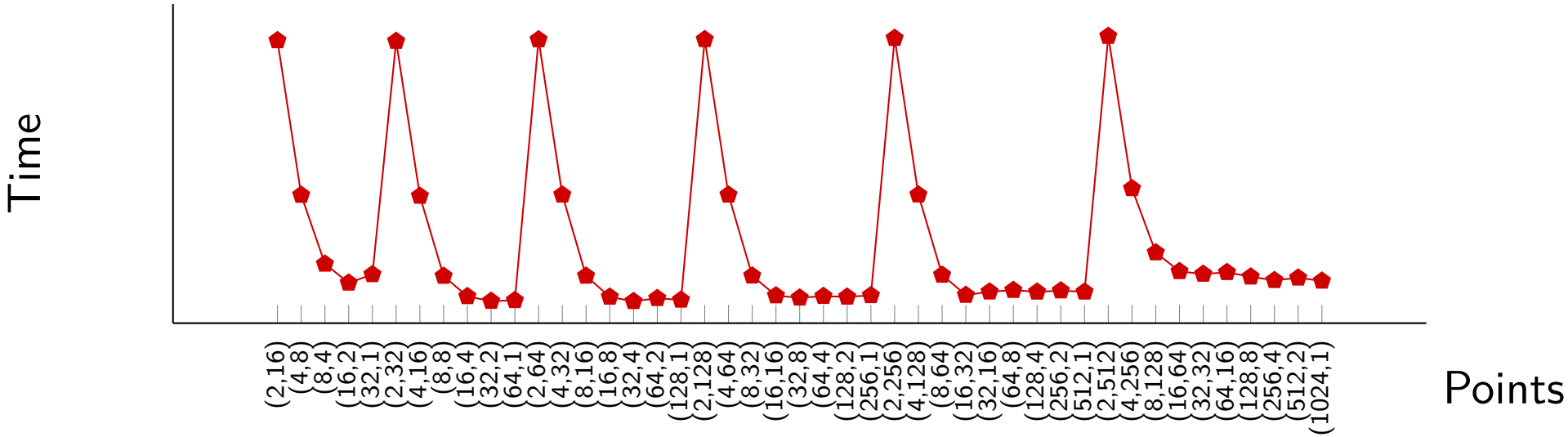


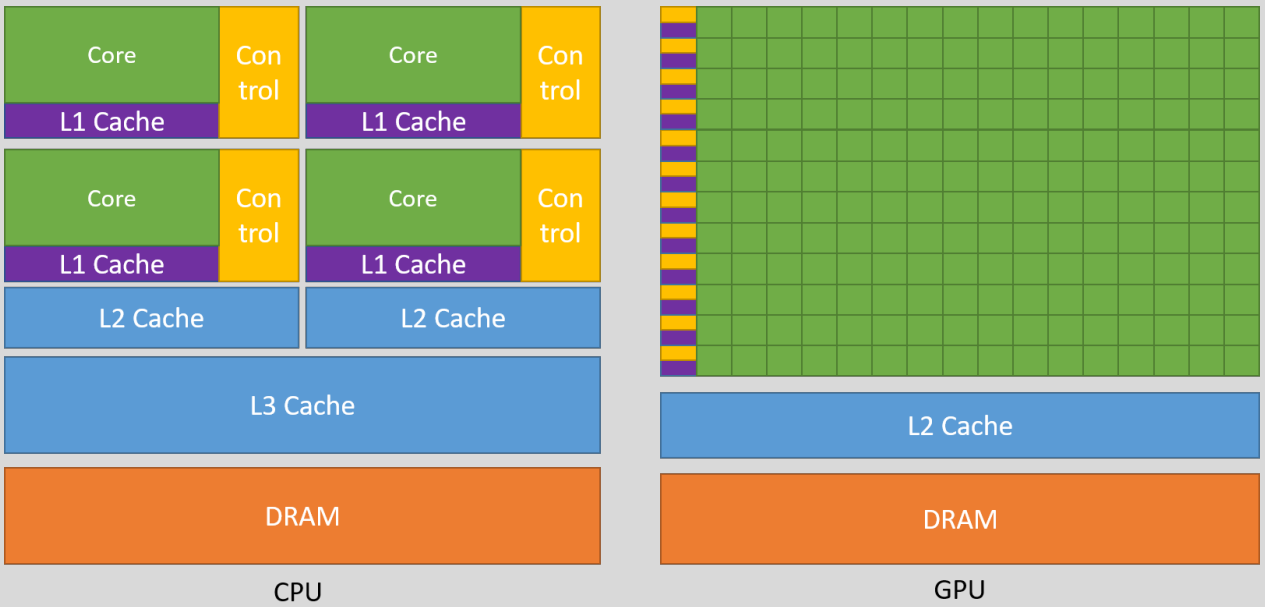
Figure: Example polybench
_2DCONV Kernel Convolution2D_kernel for input size 8192

- They depend on data size and cannot be determined at compile time.
- klaraptor's goal: automatically and dynamically find the values of the launch parameters which optimize the performance of a CUDA kernel.
- We achieve this goal for CUDA kernels implementing data-oblivious algorithms.

Outline

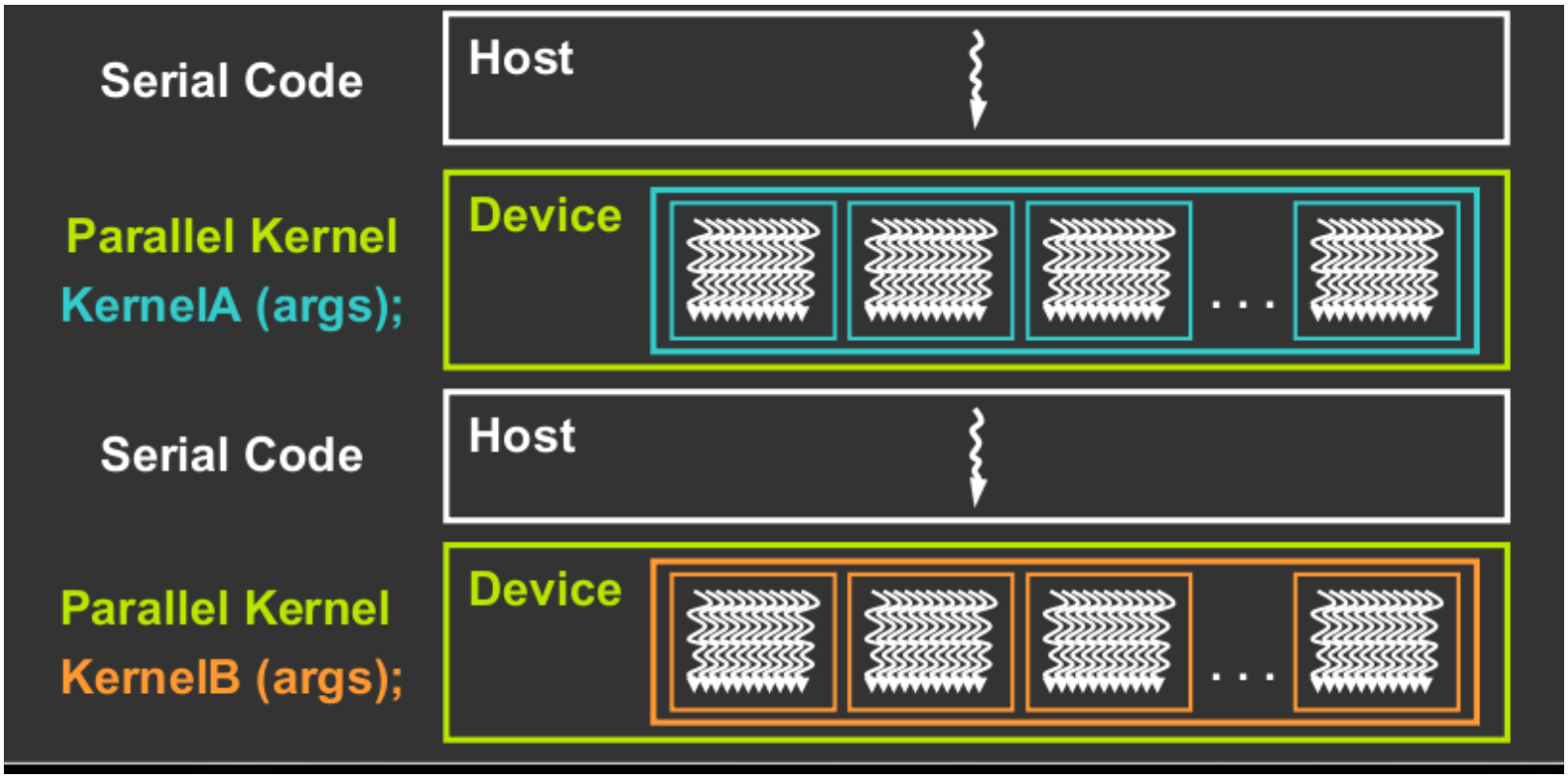
Graphics Processing Units (GPUs)

- GPUs are designed for massive parallelism, while CPUs focus on sequential processing.
- GPUs handle massive amounts of data and perform the same operation on them simultaneously (SIMD).

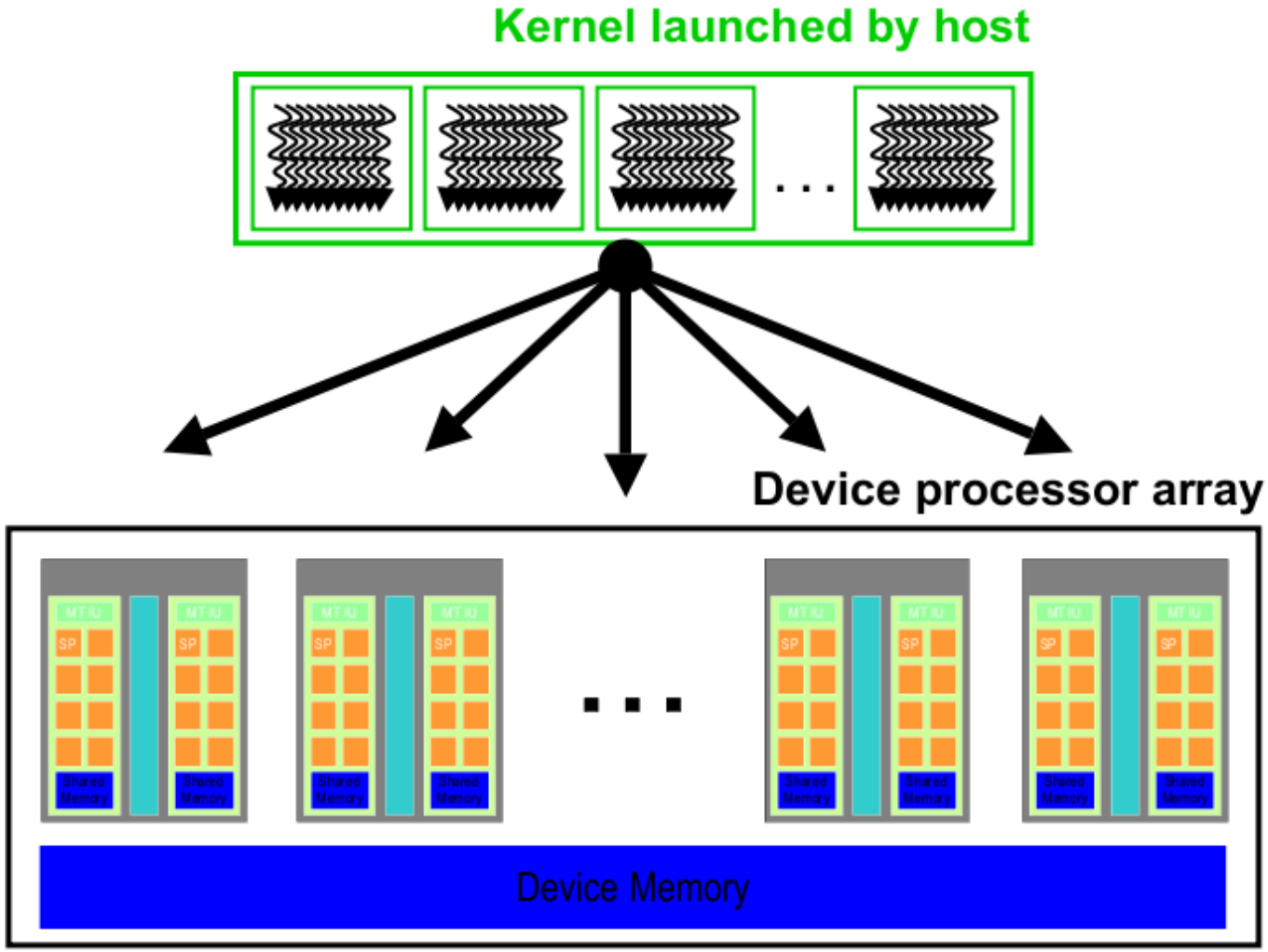


Heterogeneous programming

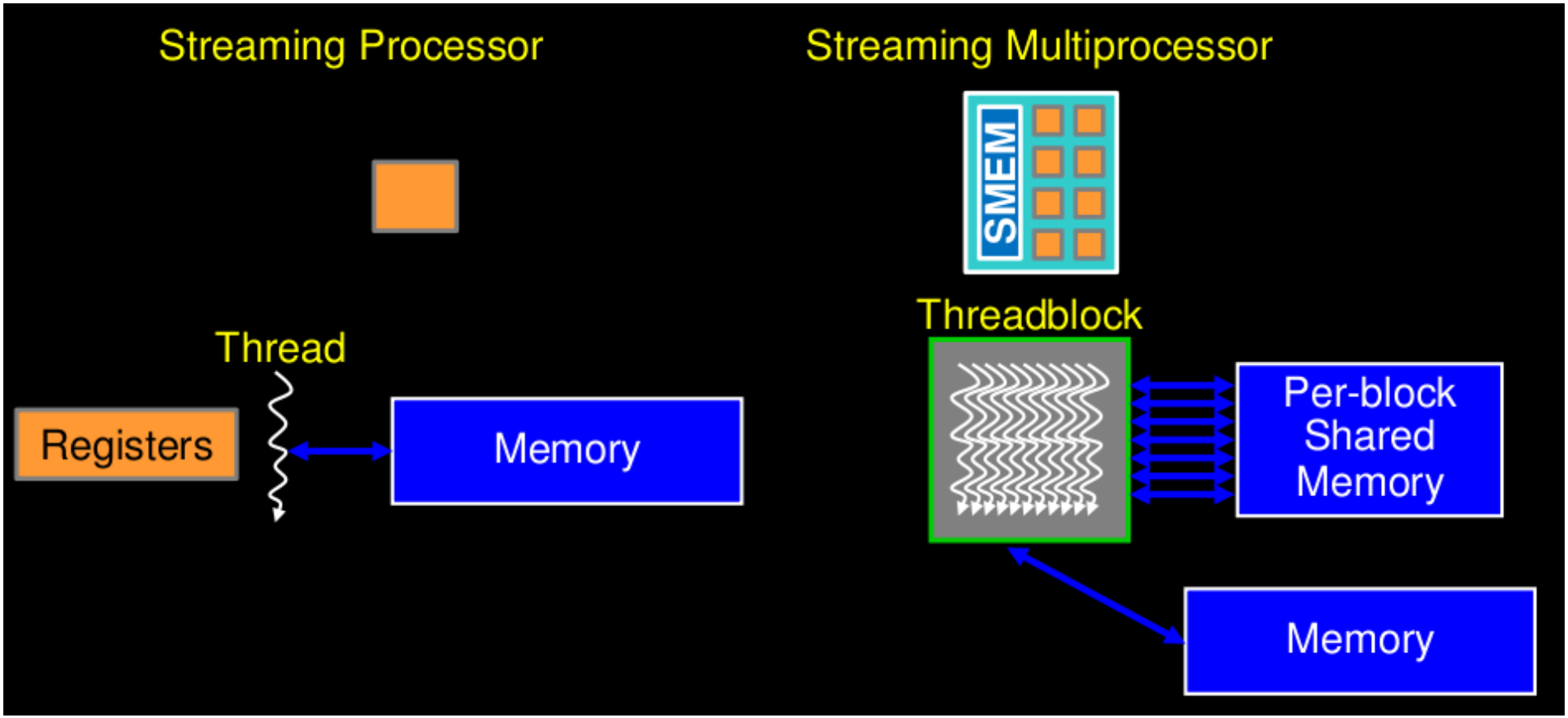
- A CUDA program is a serial program with parallel kernels, all in C.
- The serial C code executes in a host (= CPU) thread
- The parallel kernel C code executes in many device threads across multiple GPU processing elements, called streaming processors (SP).



Blocks Run on Multiprocessors



Streaming processors and multiprocessors



Example: increment array elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    increment_cpu(a, b, N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if( idx < N)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize) );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Estimating running time with the MWP-CWP model (1/2)

- Data transfer between one streaming multiprocessors (SMs) and the global memory can take 100's of clock cycles

Estimating running time with the MWP-CWP model (1/2)

- Data transfer between one streaming multiprocessors (SMs) and the global memory can take 100's of clock cycles
- SMs typically run several groups of threads (warps) concurrently so as to hide those delays with computations

Estimating running time with the MWP-CWP model (1/2)

- Data transfer between one streaming multiprocessors (SMs) and the global memory can take 100's of clock cycles
- SMs typically run several groups of threads (warps) concurrently so as to hide those delays with computations
- Two quantities measure the effectiveness of this mechanism: the Memory Warp Parallelism (MWP) and the Computation Warp Parallelism (CWP), see [14].

Estimating running time with the MWP-CWP model (1/2)

- Data transfer between one streaming multiprocessors (SMs) and the global memory can take 100's of clock cycles
- SMs typically run several groups of threads (warps) concurrently so as to hide those delays with computations
- Two quantities measure the effectiveness of this mechanism: the Memory Warp Parallelism (MWP) and the Computation Warp Parallelism (CWP), see [14].
- MWP and CWP can be determined by code profiling and cannot determined by static analysis.

Estimating running time with the MWP-CWP model (1/2)

- Data transfer between one streaming multiprocessors (SMs) and the global memory can take 100's of clock cycles
- SMs typically run several groups of threads (warps) concurrently so as to hide those delays with computations
- Two quantities measure the effectiveness of this mechanism: the Memory Warp Parallelism (MWP) and the Computation Warp Parallelism (CWP), see [14].
- MWP and CWP can be determined by code profiling and cannot determined by static analysis.
- MWP and CWP (combined with hardware characteristics) produces a sharp estimate of the running time of a (simple) kernel.

Estimating running time with the MWP-CWP model (1/2)

- Data transfer between one streaming multiprocessors (SMs) and the global memory can take 100's of clock cycles
- SMs typically run several groups of threads (warps) concurrently so as to hide those delays with computations
- Two quantities measure the effectiveness of this mechanism: the Memory Warp Parallelism (MWP) and the Computation Warp Parallelism (CWP), see [14].
- MWP and CWP can be determined by code profiling and cannot determined by static analysis.
- MWP and CWP (combined with hardware characteristics) produces a sharp estimate of the running time of a (simple) kernel.
- The formula is given by a piece-wise multivariate rational function, see [14]

Estimating running time with the MWP-CWP model (2/2)

$$Mem\_L\_Uncoal = Mem\_LD + (\#Uncoal\_per\_mw - 1) \times Departure\_del\_uncoal \quad (10)$$

$$Mem\_L\_Coal = Mem\_LD \quad (11)$$

$$Mem\_L = Mem\_L\_Uncoal \times Weight\_uncoal + Mem\_L\_Coal \times Weight\_coal \quad (12)$$

$$Weight\_uncoal = \frac{\#Uncoal\_Mem\_insts}{(\#Uncoal\_Mem\_insts + \#Coal\_Mem\_insts)} \quad (13)$$

$$Weight\_coal = \frac{\#Coal\_Mem\_insts}{(\#Coal\_Mem\_insts + \#Uncoal\_Mem\_insts)} \quad (14)$$

$$Departure\_delay = (Departure\_del\_uncoal \times \#Uncoal\_per\_mw) \times Weight\_uncoal + Departure\_del\_coal \times Weight\_coal \quad (15)$$

$$MWP\_Without\_BW\_full = Mem\_L / Departure\_delay \quad (16)$$

$$MWP\_Without\_BW = MIN(MWP\_Without\_BW\_full, \#Active\_warps\_per\_SM) \quad (17)$$

$$Mem\_cycles = Mem\_L\_Uncoal \times \#Uncoal\_Mem\_insts + Mem\_L\_Coal \times \#Coal\_Mem\_insts \quad (18)$$

$$Comp\_cycles = \#Issue\_cycles \times (\#total\_insts) \quad (19)$$

$$N = \#Active\_warps\_per\_SM \quad (20)$$

$$\#Rep = \frac{\#Blocks}{\#Active\_blocks\_per\_SM \times \#Active\_SMs} \quad (21)$$

If (MWP is N warps per SM) and (CWP is N warps per SM)

$$Exec\_cycles\_app = (Mem\_cycles + Comp\_cycles + \frac{Comp\_cycles}{\#Mem\_insts} \times (MWP - 1)) \times \#Rep \quad (22)$$

If (CWP >= MWP) or (Comp_cycles > Mem_cycles)

$$Exec\_cycles\_app = (Mem\_cycles \times \frac{N}{MWP} + \frac{Comp\_cycles}{\#Mem\_insts} \times (MWP - 1)) \times \#Rep \quad (23)$$

If (MWP > CWP)

$$Exec\_cycles\_app = (Mem\_L + Comp\_cycles \times N) \times \#Rep \quad (24)$$

Outline

Klaraptor's underlying theory (1/2)

- Let $\mathcal{P}$ be a multithreaded program implementing a data-oblivious algorithms to be executed on a specific multiprocessor.

Klaraptor's underlying theory (1/2)

- Let $\mathcal{P}$ be a multithreaded program implementing a data-oblivious algorithms to be executed on a specific multiprocessor.
- Parameters influencing the performance of $\mathcal{P}$ include:

Klaraptor's underlying theory (1/2)

- Let $\mathcal{P}$ be a multithreaded program implementing a data-oblivious algorithms to be executed on a specific multiprocessor.
- Parameters influencing the performance of $\mathcal{P}$ include:
  1. *data parameters* $D = (D_1, \ldots, D_d)$, specifying the size and possibly structural characteristics of the data,

Klaraptor's underlying theory (1/2)

- Let $\mathcal{P}$ be a multithreaded program implementing a data-oblivious algorithms to be executed on a specific multiprocessor.
- Parameters influencing the performance of $\mathcal{P}$ include:
  1. *data parameters* $D = (D_1, \ldots, D_d)$, specifying the size and possibly structural characteristics of the data,
  2. *hardware parameters*, specifying characteristics of hardware resources, and

Klaraptor's underlying theory (1/2)

- Let $\mathcal{P}$ be a multithreaded program implementing a data-oblivious algorithms to be executed on a specific multiprocessor.
- Parameters influencing the performance of $\mathcal{P}$ include:
  1. *data parameters* $D = (D_1, \ldots, D_d)$, specifying the size and possibly structural characteristics of the data,
  2. *hardware parameters*, specifying characteristics of hardware resources, and
  3. *program parameters* $P = (P_1, \ldots, P_p)$, specifying how work (e.g. threads) is mapped to hardware resources.

Klaraptor's underlying theory (1/2)

- Let $\mathcal{P}$ be a multithreaded program implementing a data-oblivious algorithms to be executed on a specific multiprocessor.
- Parameters influencing the performance of $\mathcal{P}$ include:
  1. *data parameters* $D = (D_1, \ldots, D_d)$, specifying the size and possibly structural characteristics of the data,
  2. *hardware parameters*, specifying characteristics of hardware resources, and
  3. *program parameters* $P = (P_1, \ldots, P_p)$, specifying how work (e.g. threads) is mapped to hardware resources.
- Let $\mathcal{E}$ be a high-level performance metric (e.g. running time, memory consumption) for $\mathcal{P}$ that we want to optimize.

Klaraptor's underlying theory (1/2)

- Let $\mathcal{P}$ be a multithreaded program implementing a data-oblivious algorithms to be executed on a specific multiprocessor.
- Parameters influencing the performance of $\mathcal{P}$ include:
  1. *data parameters* $D = (D_1, \ldots, D_d)$, specifying the size and possibly structural characteristics of the data,
  2. *hardware parameters*, specifying characteristics of hardware resources, and
  3. *program parameters* $P = (P_1, \ldots, P_p)$, specifying how work (e.g. threads) is mapped to hardware resources.
- Let $\mathcal{E}$ be a high-level performance metric (e.g. running time, memory consumption) for $\mathcal{P}$ that we want to optimize.
- Given values of D, our goal is to find values of the P such that the execution of $\mathcal{P}$ optimizes $\mathcal{E}$.

Klaraptor's underlying theory (2/2)

- Performance prediction models (like the MWP-CWP model) attempt to estimate $\mathcal{E}$ from a combination of P, D, and platform-specific low-level metrics $L = (L_1, \ldots, L_\ell)$ (memory throughput, cache miss rate, etc.).

Klaraptor's underlying theory (2/2)

- Performance prediction models (like the MWP-CWP model) attempt to estimate $\mathcal{E}$ from a combination of P, D, and platform-specific low-level metrics $L = (L_1, \ldots, L_\ell)$ (memory throughput, cache miss rate, etc.).
- Thanks to our data-oblivious hypothesis, We deduce that these low-level performance metrics are themselves rational functions of P, D.

Klaraptor's underlying theory (2/2)

- Performance prediction models (like the MWP-CWP model) attempt to estimate $\mathcal{E}$ from a combination of P, D, and platform-specific low-level metrics $L = (L_1, \ldots, L_\ell)$ (memory throughput, cache miss rate, etc.).
- Thanks to our data-oblivious hypothesis, We deduce that these low-level performance metrics are themselves rational functions of P, D.

To address our optimization goal, we use the following strategy.

Klaraptor's underlying theory (2/2)

- Performance prediction models (like the MWP-CWP model) attempt to estimate $\mathcal{E}$ from a combination of P, D, and platform-specific low-level metrics $L = (L_1, \ldots, L_\ell)$ (memory throughput, cache miss rate, etc.).
- Thanks to our data-oblivious hypothesis, We deduce that these low-level performance metrics are themselves rational functions of P, D.

To address our optimization goal, we use the following strategy.

- At the compile-time of program $\mathcal{P}$, for each metric, we determine (by evaluation-interpolation) a mathematical formula expressing that metric as a function of the data and program parameters.

Klaraptor's underlying theory (2/2)

- Performance prediction models (like the MWP-CWP model) attempt to estimate $\mathcal{E}$ from a combination of P, D, and platform-specific low-level metrics $L = (L_1, \ldots, L_\ell)$ (memory throughput, cache miss rate, etc.).
- Thanks to our data-oblivious hypothesis, We deduce that these low-level performance metrics are themselves rational functions of P, D.

To address our optimization goal, we use the following strategy.

- At the compile-time of program $\mathcal{P}$, for each metric, we determine (by evaluation-interpolation) a mathematical formula expressing that metric as a function of the data and program parameters.
- This mathematical formula takes the form of a piece-wise rational function.

Klaraptor's underlying theory (2/2)

- Performance prediction models (like the MWP-CWP model) attempt to estimate $\mathcal{E}$ from a combination of P, D, and platform-specific low-level metrics $L = (L_1, \ldots, L_\ell)$ (memory throughput, cache miss rate, etc.).
- Thanks to our data-oblivious hypothesis, We deduce that these low-level performance metrics are themselves rational functions of P, D.

To address our optimization goal, we use the following strategy.

- At the compile-time of program $\mathcal{P}$, for each metric, we determine (by evaluation-interpolation) a mathematical formula expressing that metric as a function of the data and program parameters.
- This mathematical formula takes the form of a piece-wise rational function.
- At the runtime of $\mathcal{P}$, given specific values of D and a choice of P, we can evaluate those functions for each metric and thus compute $\mathcal{E}$.

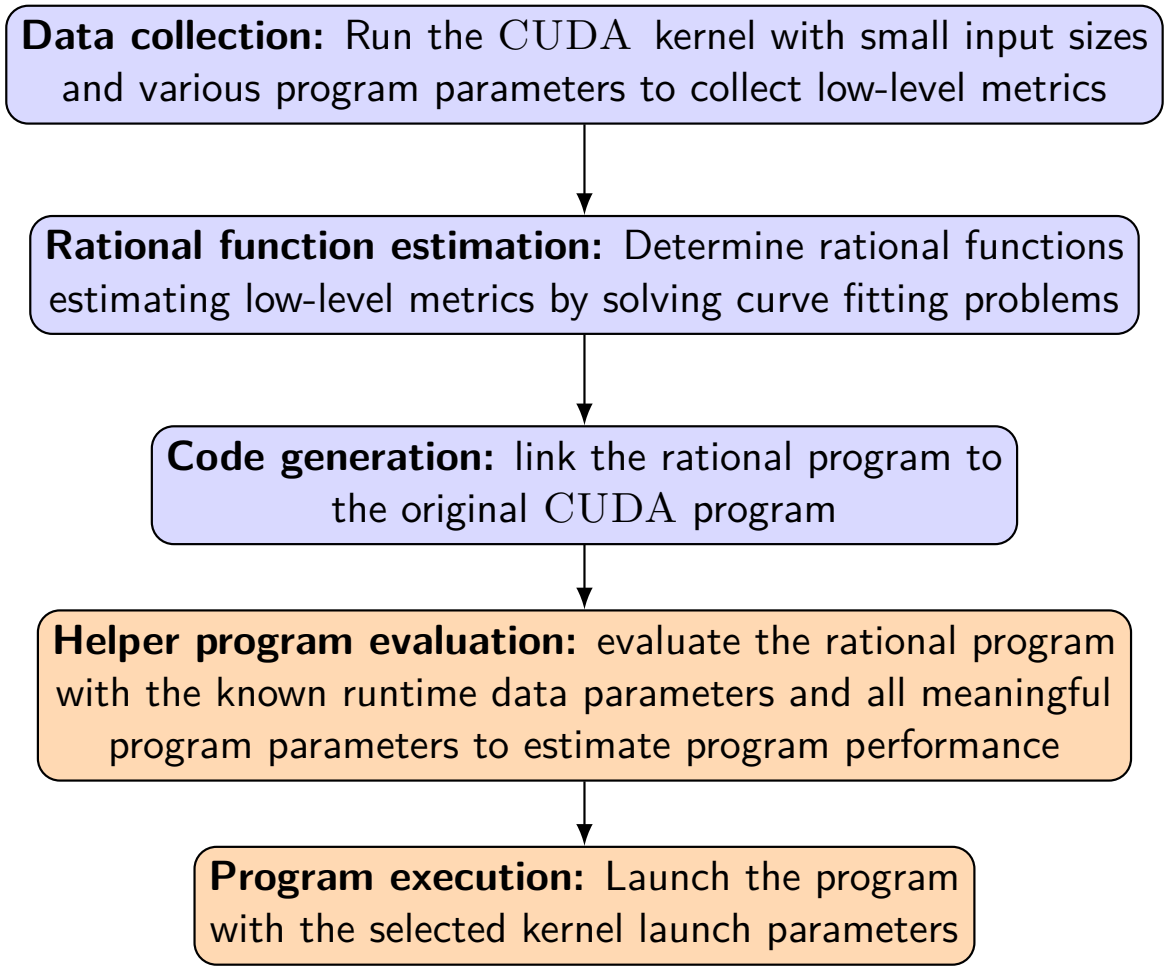
Klaraptor's underlying theory (2/2)

- Performance prediction models (like the MWP-CWP model) attempt to estimate $\mathcal{E}$ from a combination of P, D, and platform-specific low-level metrics $L = (L_1, \ldots, L_\ell)$ (memory throughput, cache miss rate, etc.).
- Thanks to our data-oblivious hypothesis, We deduce that these low-level performance metrics are themselves rational functions of P, D.

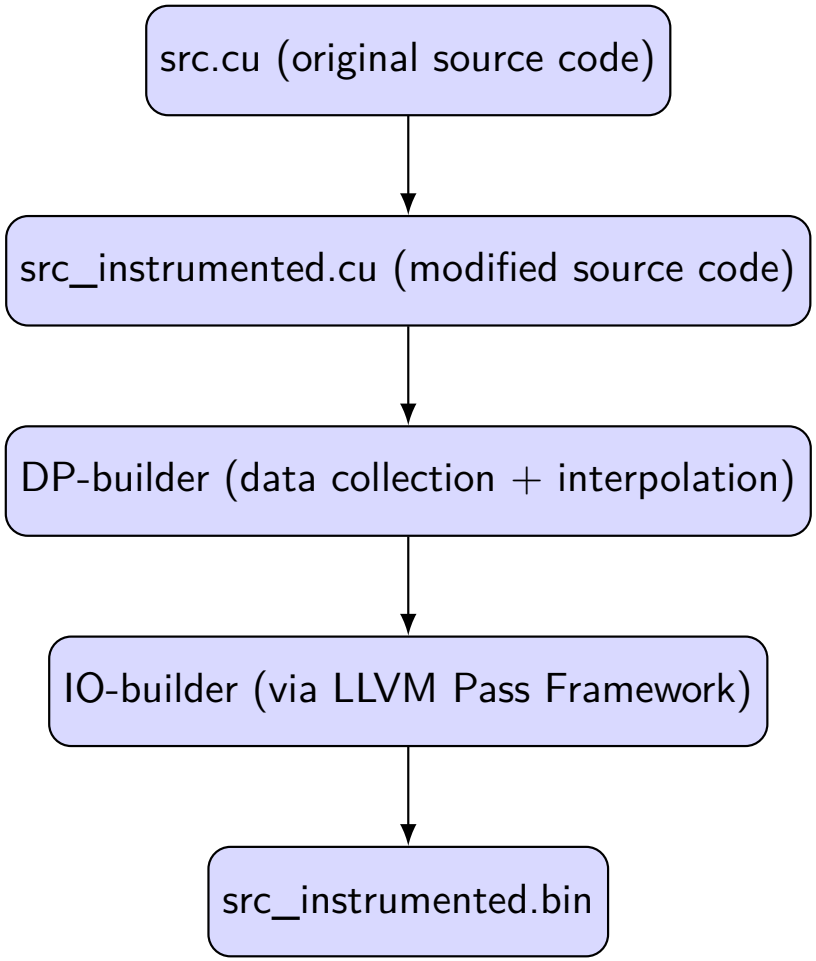
To address our optimization goal, we use the following strategy.

- At the compile-time of program $\mathcal{P}$, for each metric, we determine (by evaluation-interpolation) a mathematical formula expressing that metric as a function of the data and program parameters.
- This mathematical formula takes the form of a piece-wise rational function.
- At the runtime of $\mathcal{P}$, given specific values of D and a choice of P, we can evaluate those functions for each metric and thus compute $\mathcal{E}$.
- Repeating this for all possible choices of P (assumed to be finite in

Main steps of Klaraptor's algorithm



High-level view of Klaraptor's implementation



We have used:

- *LLVM Pass Framework* for modifying the code at the IR level
- *NVIDIA Nsight Compute CLI* to do the data collection
- *CLAPACK and ATLAS* for the numerical computations done in the curve fitting step
- system specs: LLVM 11, CUDA 11, CLAPACK, python 2.7.

Annotations and preprocessing source code

```
#pragma kernel_info_size_param_idx_Sample = 1;
#pragma kernel_info_dim_sample_kernel = 2;

__global__ void Sample (int *A, int N) {
    int tid_x = threadIdx.x + blockIdx.x*blockDim.x;
    int tid_y = threadIdx.y + blockIdx.y*blockDim.y;
    ...
}
```

- Annotating and preprocessing the source code makes it compatible with the KLARAPTOR tool, enabling the automatic determination of optimal kernel launch parameters.
- CUDA program should target at least CUDA Compute Capability 7.5, no CUDA runtime API calls, and block and grid dimensions must be declared as dim3 structs.

Outline

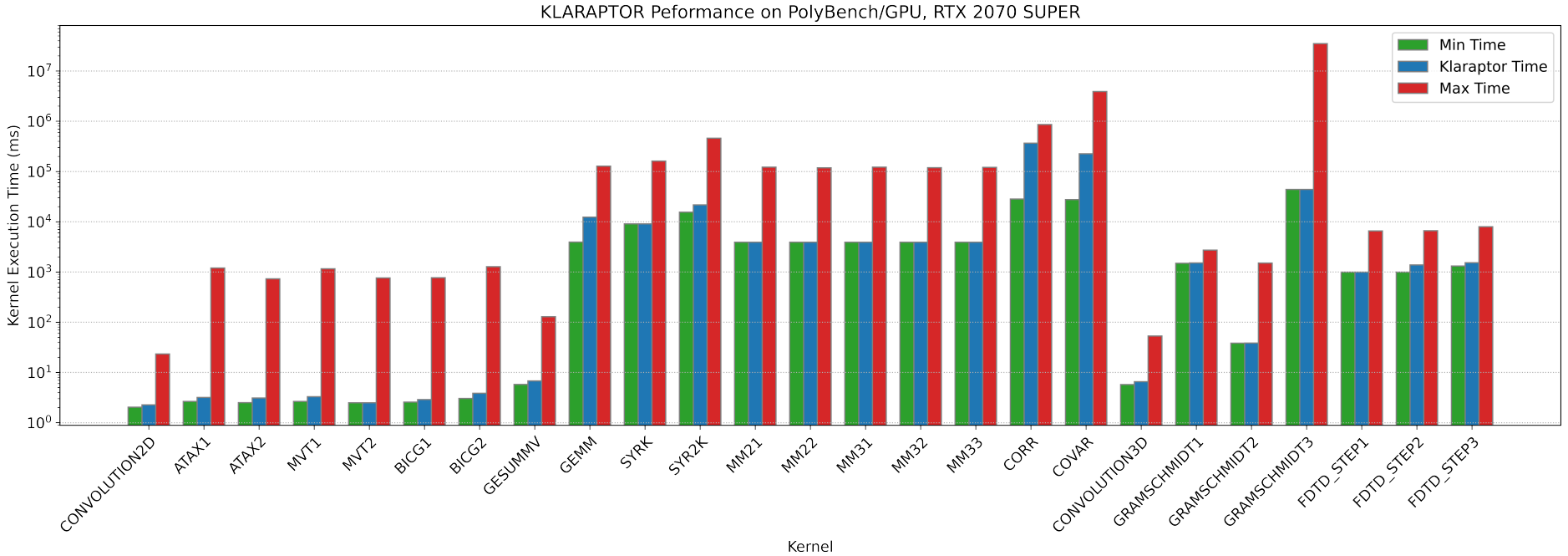


Figure: Comparing kernel execution time (log-scaled) for the thread block configuration chosen by KLARAPTOR versus the minimum and maximum times as determined by an exhaustive search over all possible configurations. Kernels are part of the `PolyBench/GPU` benchmark suite and executed on a RTX 2070 SUPER with a data size of $N = 8192$ (except convolution3d with $N = 512$)

Table: KLARAPTOR Optimization Times on `Polybench/GPU`, RTX 2070 SUPER
Comparing times for (1) compile-time optimization steps of KLARAPTOR, (2) exhaustive search over all thread block configurations, the execution time for a kernel given (3) the best thread block configuration, and (4) the worst thread block configuration. Exhaustive search is given as a sum for values up to $N = 8192$ (except convolution3d with $N = 512$).

Kernel	KLARAPTOR Time (s) $128 \leq N < \infty$	Ex. Search Time (s) $128 \leq N \leq 8192$	Min Time (s) $N = 8192$	Max Time (s) $N = 8192$
2DCONV	210.29	82.78	0.002	0.023
ATAX	507.59	59.60	0.006	1.940
MVT	508.03	60.03	0.005	1.978
BICG	510.91	60.16	0.006	2.050
GESUMMV	398.54	142.78	0.006	0.129
GEMM	456.50	987.77	3.941	126.052
SYRK	579.84	2772.64	9.069	160.944
SYR2K	1173.68	9553.64	15.534	459.169
2MM	700.49	1889.62	7.851	240.828
3MM	944.54	2798.12	11.779	361.310
CORR	1032.92	10924.12	28.365	861.289
COVAR	1141.45	23251.12	27.670	3900.855
3DCONV	132.88	52.06	0.006	0.053
GRAMSCHM	2113.27	94206.06	45.418	35146.314
FDTD_2D	489.21	495.79	3.304	21.107

Outline

This is a joint work with:

- Delaram Talaashrafi (NVIDIA)
- Johannes Doerfert (Lawrence Livermore National Laboratory)

Published in IWOMP-2022:

- D. Talaashrafi, MMMM, and J. Doerfert, “Towards automatic openmp-aware utilization of fast GPU memory," in International Workshop on OpenMP, Springer, 2022, pp. 67–80.

Outline

Conclusions

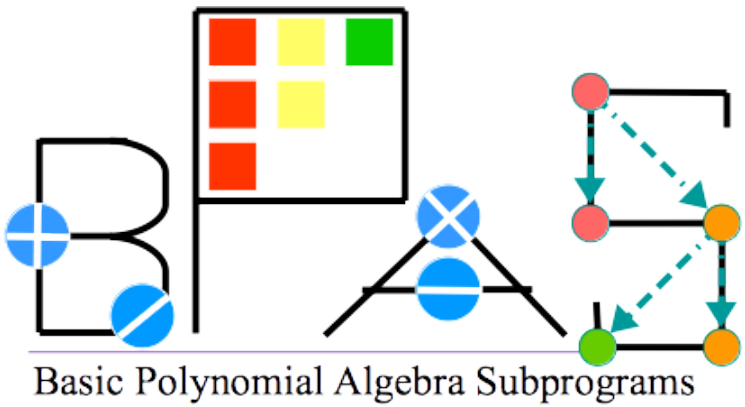
Pipelines:

- Pipelining is an underutilized pattern with lots of potential application in computer algebra
- We presented a mechanism for discovering and exploiting pipelines over a sequence of for-loop nests
- This is completely (no competitive methods) and implemented in LLVM/Polly
- Ask Delaram or Johannes if you are interested.

Klaraptor:

- Kernel launch parameters of GPU kernels are important-and-hard right
- We propose an AI-free approach which works successfully with data-oblivious algorithms
- All other approaches for that problem use AI ...
- Ask me if you are interested.

Thank You!



http://www.bpaslib.org/

References

[1] M. Asadi, A. Brandt, C. Chen, S. Covanov, F. Mansouri, D. Mohajerani, R. H. C. Moir, M. Moreno Maza, D. Talaashrafi, L. Wang, N. Xie, and Y. Xie. *Basic Polynomial Algebra Subprograms (BPAS)*. www.bpaslib.org. 2021.

[2] M. Asadi, A. Brandt, R. H. C. Moir, M. M. Maza, and Y. Xie. "On the parallelization of triangular decompositions". In: *International Symposium on Symbolic and Algebraic Computation (ISSAC '20), Kalamata, Greece, July 20-23, 2020*. ACM, 2020, pp. 22–29.

[3] M. Asadi, A. Brandt, R. H. C. Moir, and M. Moreno Maza. "Algorithms and Data Structures for Sparse Polynomial Arithmetic". In: *Mathematics* 7.5 (2019), p. 441.

[4] M. Asadi, A. Brandt, R. H. C. Moir, M. Moreno Maza, and Y. Xie. "Parallelization of Triangular Decompositions: Techniques and Implementation". In: *J. Symb. Comput.* (2021). *(to appear).*

[5] A. Brandt, R. H. C. Moir, and M. Moreno Maza. "Employing C++ Templates in the Design of a Computer Algebra Library". In: *Mathematical Software - ICMS 2020, Braunschweig, Germany, July 13-16, 2020*. Vol. 12097. LNCS. Springer, 2020, pp. 342–352.

[6] A. Brandt and M. Moreno Maza. "On the Complexity and Parallel Implementation of Hensel's Lemma and Weierstrass Preparation". In: *Computer Algebra in Scientific Computing (CASC '21)*. (*To appear*). 2021.

[7] J. Della Dora and J. Fitch, eds. *Computer Algebra and Parallelism, First International Workshop, Grenoble, France, September 1989*. Academic Press, 1988. ISBN: 978-0122090424.

[8] J. Dumas, E. L. Kaltofen, and C. Pernet, eds. *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation, PASCO 2015, Bath, United Kingdom, July 10-12, 2015*. ACM, 2015. ISBN: 978-1-4503-3599-7.

[9] J. Faugère, M. B. Monagan, and H. Loidl, eds. *Proceedings of the 2017 International Workshop on Parallel Symbolic Computation, PASCO 2017, Kaiserslautern, Germany, July 23-24, 2017*. ACM, 2017. ISBN: 978-1-4503-5288-8.

[10] M. Gastineau and J. Laskar. "Highly Scalable Multiplication for Distributed Sparse Multivariate Polynomials on Many-Core Systems". In: *Computer Algebra in Scientific Computing (CASC '13)*. Vol. 8136. LNCS. Springer, 2013, pp. 100–115.

[11] M. Gastineau and J. Laskar. "Parallel sparse multivariate polynomial division". In: *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation, PASCO 2015*. ACM, 2015, pp. 25–33.

[12] H. Hong, ed. *Proceedings of the 1st International Workshop on Parallel Symbolic Computation, PASCO 1994, Linz, Austria*. World scientific, 1994. ISBN: 978-9810220402.

[13] H. Hong, E. Kaltofen, and M. A. Hitz, eds. *Proceedings of the 2nd International Workshop on Parallel Symbolic Computation, PASCO 1997, July 20-22, 1997, Kihei, Hawaii, USA*. ACM, 1997. ISBN: 0-89791-951-3.

[14] S. Hong and H. Kim. "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness". In: *Proc. of ISCA*. 2009, pp. 152–163.

[15] K. Kourtis, M. Dazzi, N. Ioannou, T. Grosser, A. Sebastian, and E. Eleftheriou. "Compiling Neural Networks for a Computational Memory Accelerator". In: (2020).

[16] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[17] M. B. Monagan and R. Pearce. "Parallel sparse polynomial multiplication using heaps". In: *International Symposium on Symbolic and Algebraic Computation, (ISSAC '09), Seoul, Republic of Korea, July 29-31, 2009*. ACM, 2009.

[18] M. B. Monagan and R. Pearce. "Sparse polynomial division using a heap". In: *J. Symb. Comput.* 46.7 (2011), pp. 807–822.

[19] M. Moreno Maza and J. Roch, eds. *Proceedings of the 4th International Workshop on Parallel Symbolic Computation, PASCO 2010, July 21-23, 2010, Grenoble, France*. ACM, 2010. ISBN: 978-1-4503-0067-4.

[20] M. Moreno Maza and S. M. Watt, eds. *Proceedings of the 3rd International Workshop on Parallel Symbolic Computation, PASCO 2007, 27-28 July 2007, London, Ontario, Canada*. ACM, 2007. ISBN: 978-1-59593-741-4.

[21] M. Moreno Maza and Y. Xie. "Balanced Dense Polynomial Multiplication on Multi-Cores". In: *Int. J. Found. Comput. Sci.* 22.5 (2011), pp. 1035–1055.

[22] H. Razanajato, C. Bastoul, and V. Loechner. "Pipelined Multithreading Generation in a Polyhedral Compiler". In: *IMPACT 2020, in conjunction with HiPEAC 2020*. 2020.

[23] R. Zippel, ed. *Computer Algebra and Parallelism, Second International Workshop, Ithaca, USA, May 9-11, 1990*. Vol. 584. LNCS. Springer, 1992. ISBN: 3-540-55328-2.