Models of Computation for Graphics Processing Units

Marc Moreno Maza

University of Western Ontario

November 5, 2017

Contents

Overview

Models of computation for algorithm analysis

- Parallel random access machine (PRAM) model
- Fork-join model
- Threaded many-core memory (TMM) model
- The Many-core-Machine (MCM) model

Models of computation for program analysis

- The Memory-level and Thread-level Parallelism (MWP-CWP) Model

Handling machine and program parameters

Plan

Overview

Models of computation for algorithm analysis

Models of computation for program analysis

Handling machine and program parameters

Models of computation for GPUs

Background

- Analyzing, optimizing GPU code amd knowing what to expect is hard and tiem consuming, even for experts.
- This, among other reasons, stimulates the developemnt of tools generating GPU code from annoated C/C++ code
- Thus, tools for annalyzing for both algorithms and code targeting GPUs are needed.

Program models vs algorithm models

- Program models (like the MWP-CWP model) require the availability of machine-like code while algorithm models (like TMM, MCM) do not.
- On the other hand, program models give performance estimates (running time, memory consumption, etc.) which are more precise than those provided by algorithm models.

Challenges in designing a model of computation

Theoretical aspects

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

Practical aspects

- C-to-CUDA type of tools need to treat program parameters unknown symbols for portability and performenace reasons
- Analyzing parametric programs (with unknown machine and program parameters) require symbolic computation.

Plan

Overview

Models of computation for algorithm analysis

- Parallel random access machine (PRAM) model
- Fork-join model
- Threaded many-core memory (TMM) model
- The Many-core-Machine (MCM) model

Models of computation for program analysis

Handling machine and program parameters

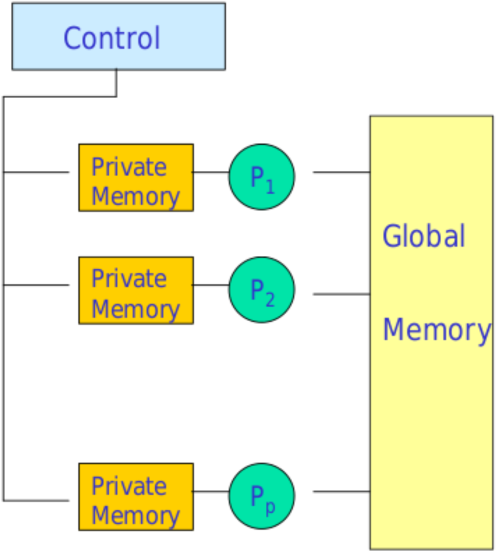
The PRAM Model: basics

Architecture

The *Parallel Random Access Machine* is a natural generalization of RAM. It is also an idealization of a *shared memory machine*. Its features are as follows.

- It holds an *unbounded collection of RAM processors* $P_0, P_1, P_2, \ldots$ **without tapes**.
- It holds an *unbounded collection of shared memory cells* $M[0], M[1], M[2], \ldots$
- Each processor P_i has its own (unbounded) local memory (register set) and P_i knows its index i.
- Each processor P_i can access any shared memory cell $M[j]$ in *unit time*, unless there is a conflict (see further).

The PRAM Model: basics



The PRAM Model: basics

Program execution

- The input of a PRAM program consists of n items stored in $M[0], \ldots, M[n-1]$.
- The output of a PRAM program consists of n' items stored in n' memory cells, say $M[n], \ldots, M[n+n'-1]$.
- A PRAM instruction executes in a 3-phase cycle:
  1. **Read** (if needed) from a shared memory cell,
  2. **Compute** locally (if needed),
  3. **Write** in a shared memory cell (if needed).
- All processors execute their 3-phase cycles **synchronously**.
- **Special assumptions** have to be made in order to resolve shared memory access conflicts.
- The only way processors can exchange data is by writing into and reading from memory cells.

The PRAM Model: complexity measures

Summary of main assumptions

- Inputs/Outputs are placed in the global memory
- Memory cell stores an arbitrarily large integer
- Each instruction takes unit time
- Instructions are synchronized across the processors

PRAM complexity measures

time: time taken by the longest running processor

space: number of memory cells accessed

proc: maximum number of active processors

The PRAM Model: code example

Computing the maximum of n numbers in $O(1)$

Input: n integer numbers stored in $M[1], \ldots, M[n]$, where $n \geq 2$.

Output: The maximum of those numbers, written at $M[n+1]$.

Program:

```
Active Proocessors P[1], ...,P[n^2];
// id the index of one of the active processor
if (id <= n)
    M[n + id] := true;
i := ((id -1) mod n) + 1;
j := ((id -1) quo n) + 1;
if (M[i] < M[j])
     M[n + i] := false;
if (M[n + i] = true)
     M[n+1] := M[i];
```

The PRAM Model: remarks

Advantages

The PRAM Model is attractive for designing parallel algorithms:

- It is natural: the number of operations executed per one cycle on p processors is at most p.
- It is strong: any processor can read or write any shared memory cell in unit time.
- It is simple: ignoring any communication or synchronization overhead.

Limitations towards GPU implementation

The PRAM model:

- treats uniformly all costs (computations, memory accesses)
- Is based on a single parallelism scheme whereas CUDA combines SIMD and the fork-join model

Hybrid CPU-GPU system

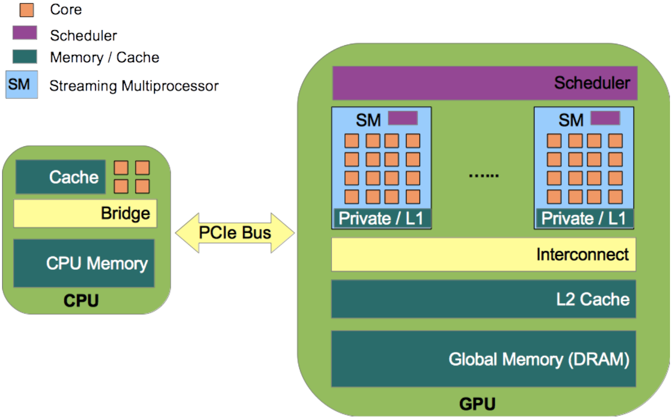
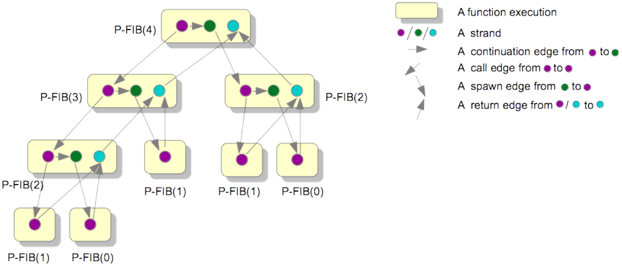


Figure: Overview of a hybrid CPU-GPU system

Fork-join model

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

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



Threaded many-core memory (TMM) model

Ma, Agrawal and Chamberlain (2014) introduce the TMM model which retains many important characteristics of GPU-type architectures.

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

Table: Machine parameters of the TMM model

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

The MCM model

(S. A, Haque, N, Xie, M., 2013) proposes a many-core machine (MCM) model which aims at

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

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

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

Characteristics of the abstract many-core machines (1/2)

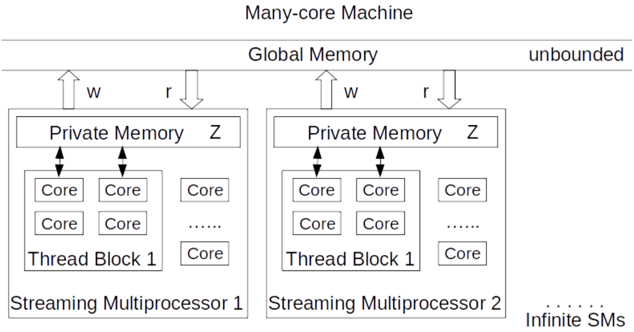


Figure: A many-core machine

- It has a global memory with high latency, while private memories have low latency.

Characteristics of the abstract many-core machines (2/2)

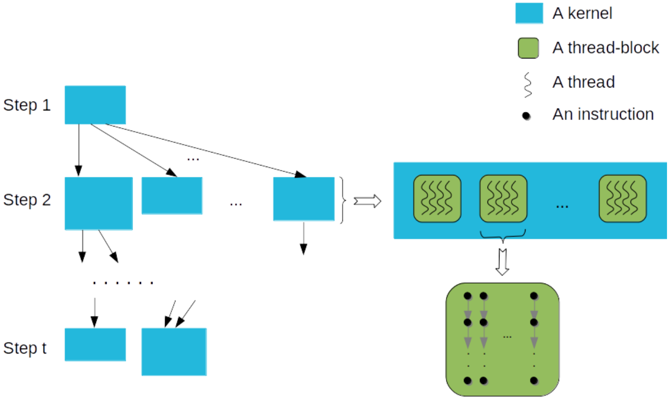


Figure: Overview of a many-core machine program, also called *kernel DAG*

Machine parameters and complexity measures

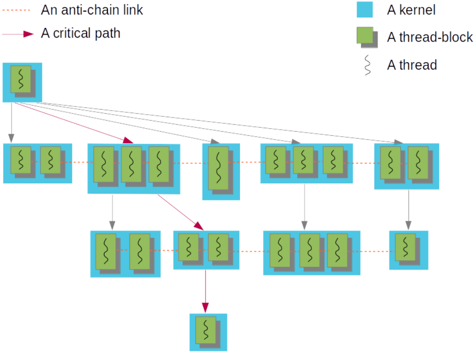
Machine parameters

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

Complexity measures

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

Characteristic quantities of the thread-block DAG



A Graham-Brent Theorem

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

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

Let K be the maximum number of thread-blocks along an anti-chain

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

Plan

Overview

Models of computation for algorithm analysis

Models of computation for program analysis

- The Memory-level and Thread-level Parallelism (MWP-CWP) Model

Handling machine and program parameters

Motivations

Limitations of previous works

- The PRAM-like models (TMM, MCM) support mainly *worst-case analysis* of algorithms targeting GPUs
- In fact those models compute T_{exec} as max(T_{local}, T_{mem} or $T_{\mathrm{local}} + T_{\mathrm{mem}}$.
- As for programs, metrics like *occupancy* are not sufficient to improve running time since they focus on a precise feature.

Goals

The MWP-CWP model (Sunpyo Hong & Hyesoon Kim, ISCA 2009)

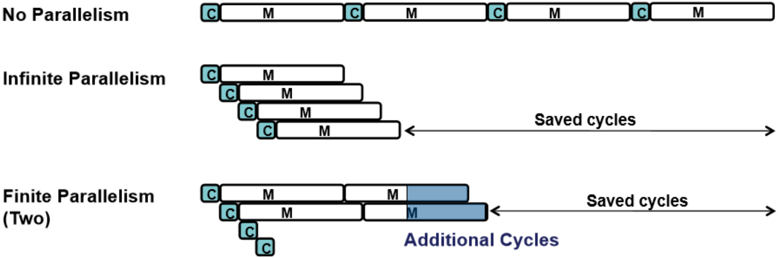
- aims at estimating T_{exec} as

$$T_{\mathrm{exec}} = T_{\mathrm{comp}} + T_{\mathrm{mem}} - T_{\mathrm{overlap}}$$

- determining T_{overlap} requires to understand whether computations hide memory latency or not
- thus requires hardware characteristics and instruction counts, thus access to the IR of a program.

Main observation of MWP-CWP

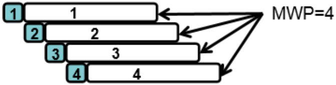
As we know, memory accesses can be overlapped between warps



Performance can be predicted by knowing the amount of *memory-level parallelism*.

Memory Warp Parallelism (MWP)

MPW is the maximum number of warps that can overlap memory accesses.

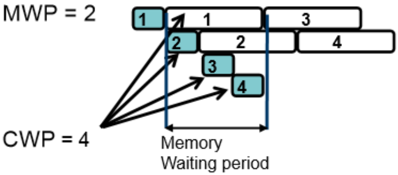


Four warps are overlapped during memory accesses

- Here, we $\mathrm{MWP} = 4$.
- MWP is determined by #Active SMs, #Active warps, Bandwidth, Types of memory accesses (Coalesced, Uncoalesced)

Computation Warp Parallelism (CWP)

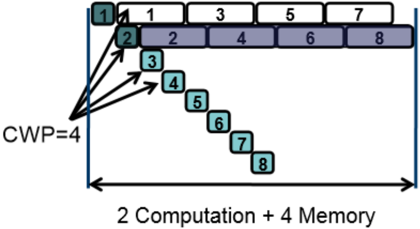
CWP is the number of warps that execute instructions during one memory access period plus one.



Here, we $\mathrm{CWP} = 4$.

Estimating the number of cycles of a kernel (1/2)

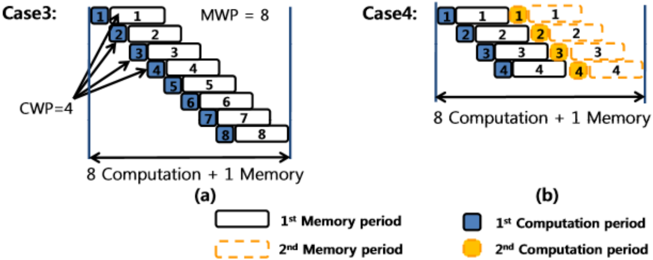
$MWP \leq CWP$



- Computation cycles are hidden by memory waiting periods
- Overall performance is dominated by the memory cycles

Estimating the number of cycles of a kernel (2/2)

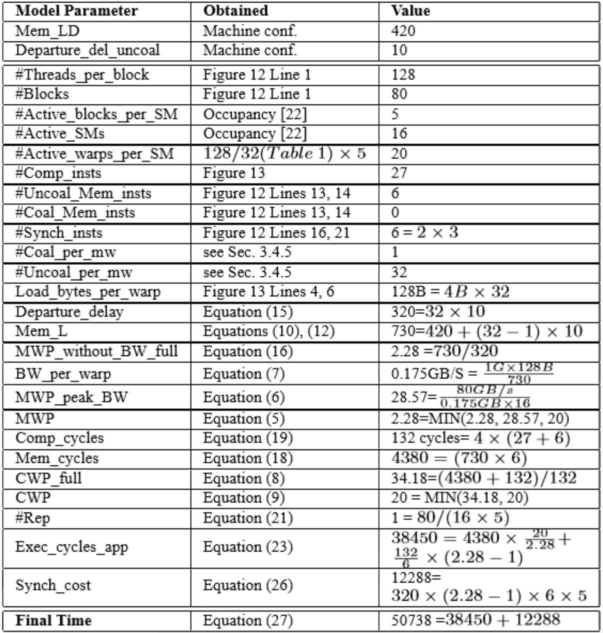
MWP > CWP



- Memory accesses are mostly hidden due to high MWP
- Overall performance is dominated by the computation cycles

See also (Jaewoong Sim & Aniruddha Dasgupta & Hyesoon Kim & Richard Vuduc, PPoPP 12)

Determining MWP and CWP



Model Parameter	Obtained	Value
Mem_LD	Machine conf.	420
Departure_del_uncoal	Machine conf.	10
#Threads_per_block	Figure 12 Line 1	128
#Blocks	Figure 12 Line 1	80
#Active_blocks_per_SM	Occupancy [22]	5
#Active_SMs	Occupancy [22]	16
#Active_warps_per_SM	$128/32(Table\ 1) \times 5$	20
#Comp_insts	Figure 13	27
#Uncoal_Mem_insts	Figure 12 Lines 13, 14	6
#Coal_Mem_insts	Figure 12 Lines 13, 14	0
#Synch_insts	Figure 12 Lines 16, 21	6 = 2×3
#Coal_per_mw	see Sec. 3.4.5	1
#Uncoal_per_mw	see Sec. 3.4.5	32
Load_bytes_per_warp	Figure 13 Lines 4, 6	128B = $4B \times 32$
Departure_delay	Equation (15)	320=32×10
Mem_L	Equations (10), (12)	730=$420 + (32 - 1) \times 10$
MWP_without_BW_full	Equation (16)	2.28 =$730/320$
BW_per_warp	Equation (7)	0.175GB/S = $\frac{1G \times 128B}{730}$
MWP_peak_BW	Equation (6)	28.57=$\frac{80GB/s}{0.175GB \times 16}$
MWP	Equation (5)	2.28=MIN(2.28, 28.57, 20)
Comp_cycles	Equation (19)	132 cycles= $4 \times (27 + 6)$
Mem_cycles	Equation (18)	$4380 = (730 \times 6)$
CWP_full	Equation (8)	34.18=$(4380 + 132)/132$
CWP	Equation (9)	20 = MIN(34.18, 20)
#Rep	Equation (21)	1 = $80/(16 \times 5)$
Exec_cycles_app	Equation (23)	$38450 = 4380 \times \frac{20}{2.28} + \frac{132}{6} \times (2.28 - 1)$
Synch_cost	Equation (26)	12288= $320 \times (2.28 - 1) \times 6 \times 5$
Final Time	Equation (27)	50738 =$38450 + 12288$

Concluding remarks

- First analytic model that estimates the execution cycles for GPU
- Experimentally, quite successful
- But, of course, requires PTX-like code and the actual hardware characteristics.

Plan

Overview

Models of computation for algorithm analysis

Models of computation for program analysis

Handling machine and program parameters

Parametric kernels

Parametric kernels

- In C-to-CUDA code generation, treating machine and program parameters improves portability and provides opportunities for better performance
- To do, we use an intermediate (high-level) language called `MetaFork`.

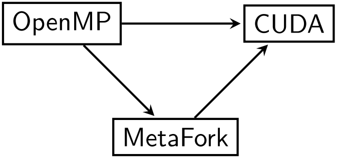
Optimizing parametric kernels

- It is desirable to optimize such parametric programs depending on the possible values of the machine and program parameters
- This generates a case discussion where each leaf consists of parametric program together with semi-algebraic constraints.

Generating *parametric* CUDA *kernels* through MetaFork permits to treat hardware and program parameters as symbols.

OpenMP:

```
#pragma omp target map(from:  d[0:N]) map(to:  c[0:N])
#pragma omp teams num_teams(T) thread_limit(B)
#pragma omp distribute parallel for schedule (static, 1)
            for (int i = 0; i < N-2; ++i)
                 d[i+1] = (c[i] + c[i+1] + c[i+2]) / 3;
```



MetaFork:

```
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;
            d[p+1] = (c[p] + c[p+1] + c[p+2])/3;
         }
   }
}
```

CUDA:

```
__global__ void kernel0(int *c, int *d, int N, int tmp5, int
tmp14, int tmp3, int tmp10) {
int b0 = blockIdx.x;
int t0 = threadIdx.x;
int private_tmp11;
int private_tmp13;
int private_tmp4;
int private_tmp9;
int private_tmp8;
int private_tmp1;
int private_i;
for (int c0 = b0; c0 < tmp5; c0 += 32768) {
   for (private_tmp11 = (t0); private_tmp11 < (tmp10);
private_tmp11 += (tmp14)) {
      for (private_tmp13 = 0; private_tmp13 < 1; private_tmp13
+= 1) {
         private_tmp4 = ((c0) * (tmp3)) + 0;
         private_tmp9 = (private_tmp11 * 1) + 0;
         private_tmp8 = (private_tmp13 * 1) + private_tmp9;
         private_tmp1 = (private_tmp8 * 1) + private_tmp4;
         private_i = (private_tmp1 * 1) + 0;
         d[private_i + 1] = (((c[private_i] + c[private_i + 1])
+ c[private_i + 2]) / 3);
      }
   }
__syncthreads();
}
}
```

Combining the LLVM infrastructure and the computer algebra system MAPLE, we implemented an algorithm optimizing programs depending on symbolic hardware and program parameters.

First case

$$\begin{cases} 2s\mathbf{B} + 2 \leq Z_B \\ 9 \leq R_B \end{cases}$$

```
for (int t = 0; t < T; ++t)
 meta_schedule cache(a) {
  meta_for (int i = 0; i < dim; i++)
   meta_for (int j = 0; j < B; j++)
    for (int k = 0; k < s; ++k) {
     int p = j+(i*s+k)*B;
     int t16 = p+1;
     int t15 = p+2;
     int p1 = t16;
     int p2 = t15;
     int np = N+p;
     int np1 = N+t16;
     int np2 = N+t15;
     if (t % 2)
      a[p1] = (a[np]+a[np1]+a[np2])/3;
     else
      a[np1] = (a[p]+a[p1]+a[p2])/3;
   }
  }
```

Second case

$$\begin{cases} 2\mathbf{B} + 2 \leq Z_B < 2s\mathbf{B} + 2 \\ 9 \leq R_B \end{cases}$$

```
for (int t = 0; t < T; ++t)
 meta_schedule cache(a) {
  meta_for (int i = 0; i < dim; i++)
   meta_for (int j = 0; j < B; j++) {
    int p = i*B+j;
    int t20 = p+1;
    int t19 = p+2;
    int p1 = t20;
    int p2 = t19;
    int np = N+p;
    int np2 = N+t19;
    int np1 = N+t20;
    if (t % 2)
     a[p1] = (a[np]+a[np1]+a[np2])/3;
    else
     a[np1] = (a[p]+a[p1]+a[p2])/3;
  }
 }
```

Third case

$$\begin{cases} Z_B < 2\mathbf{B} + 2 \\ 9 \leq R_B \end{cases}$$

```
for (int t = 0; t < T; ++t)
 meta_schedule {
  meta_for (int i = 0; i < dim; i++)
   meta_for (int j = 0; j < B; j++) {
    int p = j+i*B;
    int t16 = p+1;
    int t15 = p+2;
    int p1 = t16;
    int p2 = t15;
    int np = N+p;
    int np1 = N+t16;
    int np2 = N+t15;
    if (t % 2)
     a[p1] = (a[np]+a[np1]+a[np2])/3;
    else
     a[np1] = (a[p]+a[p1]+a[p2])/3;
   }
 }
```

Machine and program parameters

We use the same machine parameters as the MWP-CWP model

- R_B register per threads
- Z_B shared mem per block
- T_B maximum number of threads per block
- B_{SM}, maximum number of blocks per SM
- W_{SM}, maximum number of warps per SM
- SM, the number of SMs on the device.
- U time for one memory transaction between global and local memories.

Our usual program parameters are

- s the number of coefficients computed by one thread
- $B_0, B_1, \ldots$ thread-block dimension size

Optimizing parametric kernels

Estimating computing resources

- As in the MWP-CWP model, we need to estimate register pressure and shared memory usage
- The presence of symbolic parameters make the second one harder to estimate

Estimating register pressure

- It cam be estimated by an IR (say LLVM IR) of the current `MetaFork` being optimized
- Indeed, the number of registers required by a warp can be determined, even with parameters around and the result remains numerical.

Estimating shared memory usage

- Consider an array reference a[i] where *i* is a polynomial expression in the program parameters, loop counters and possibly other “intermediate variables”
- Techniques from quantifier elimination (QE) can determine the range of values of i.
- In most practical cases, efficient algorithms avoid the recourse to a general QE engine.

```
meta_for (int i = 0; i < n_block; i++)
     meta_for (int j = 0; j < n_thread; j++)
               for (int k = 0; k < s; ++k)
               {
                       int x = i * n_thread * s + k * n_thread + j;
                       int y = N - 1 - x;
                       c[y] = a[x];
            }
  }
```

The value range of x is

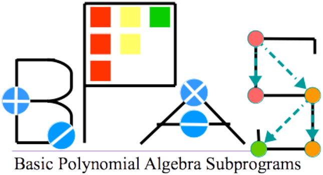
|i n_thread s, i n_thread s + (s - 1) n_thread + n_thread - 1

and the difference between end-points is n_thread s.

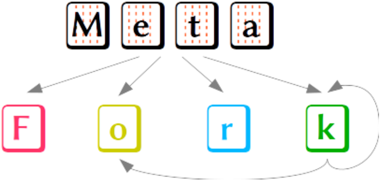
Concluding remarks

-
- It follows from the above discussion that metrics like occupancy can be determined even in the presence of parameters
- Determining MWP and CWP in the presence of parameters is work in progress
- See the demo of Haoze Yuan at the CASCON EXPO after this workshop!

Research projects with publicly available software

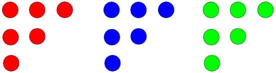


www.bpaslib.org



www.metafork.org

www.cumodp.org



www.regularchains.org

Current students

PDF & MSc: Masoud Ataei,

PhD: Mohammadali Asadi, Egor Chesakov, Ruijuan Jing, Steven Thornton, Davood Mohajerani, Robert Moir, Mehdi Samadieh

MSc: Alexander Brandt, Colin Costello, Yiming Guan, Delaram TalaAshrafi, Amha Tsegaye, Linxiao Wang,

Undergrad: Haoyu Gu, Yuchen Wang