Dynamically Finding Optimal Kernel Launch Parameters for CUDA Programs

Taabish Jeshani

ORCCA, University of Western Ontario, Canada

April 26, 2023

Co-Authorship Statement

KLARAPTOR is a joint project with Alexander Brandt, Marc Moreno Maza, Davood Mohajerani and Linxiao Wang. A preprint of this work is accessible as [1]. The contributions of the thesis author include: upgrading the profiler to support newer Nvidia GPU architectures, introduction of an outlier removal algorithm, and improvements to the overall integration of the theory of rational programs and the MWP-CWP model.

Outline

Outline

CUDA program structure

CUDA launch parameters

```
//a CUDA kernel for vector addition
__global__ void vector_addition(int *A, int *B, size_t n) {
    A[threadIdx.x] += B[threadIdx.x];
}
int main(){
    ...
    //launch the kernel with 1 block and n threads per block
    vector_addition<<<1, n>>>(Ad, Bd, n);
    ...
}
```

Grids and blocks can be of different shapes

```
dim3 block(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
dim3 grid((size_t)ceil(((float)NI)/((float)block.x)),
          (size_t)ceil(((float)NJ)/((float)block.y)));
Convolution2D_kernel<<<grid,block>>>(d_a,d_b);
```

- Launch parameters greatly impacts the performance of a given kernel
- Determining the optimal values of launch parameters for a given configuration of hardware and data parameters is critical.

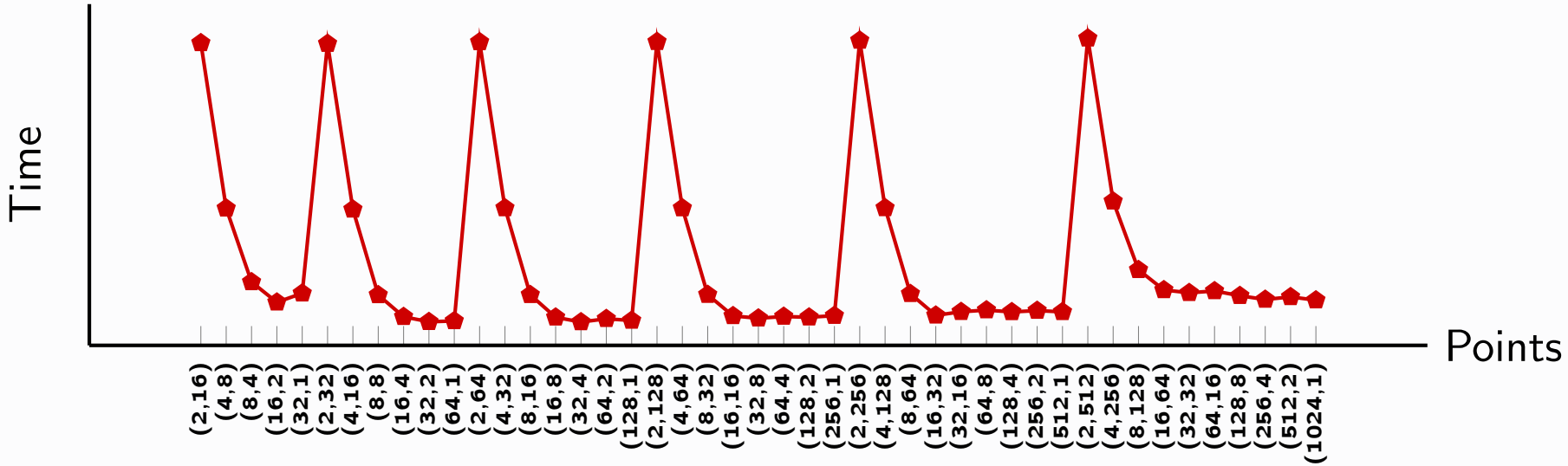


Figure: Example polybench _2DCONV Kernel Convolution2D_kernel for input size 8192

Outline

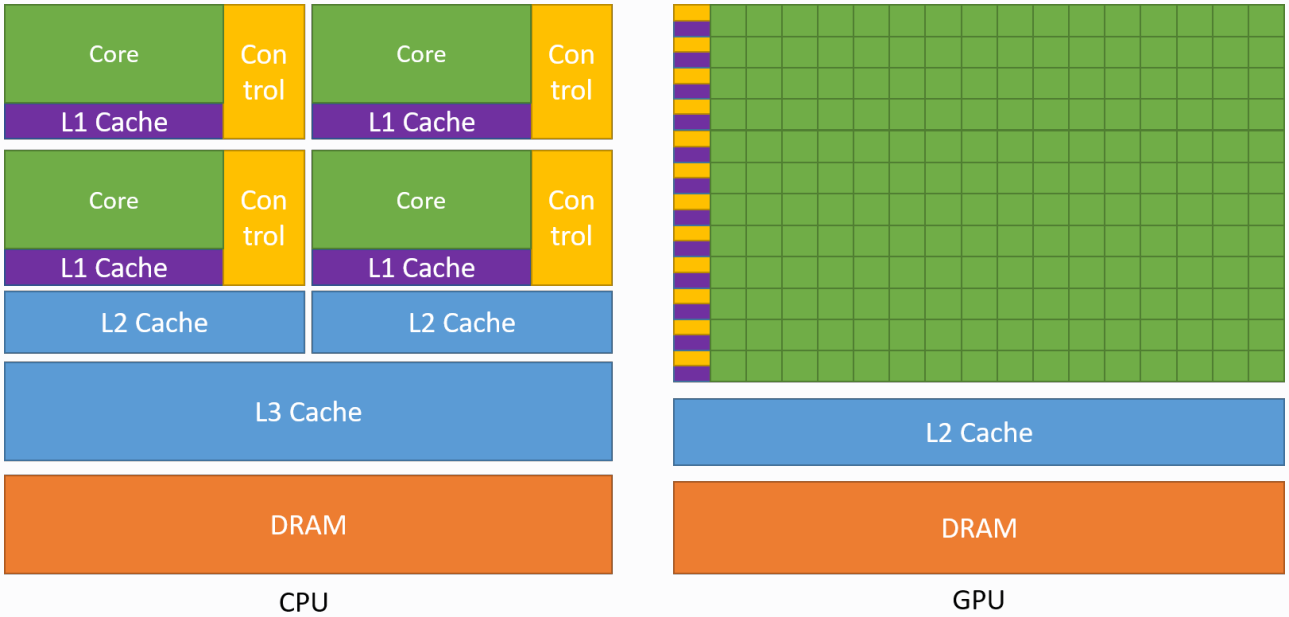
Outline

Compute Unified Device Architecture

- CUDA is a parallel programming model developed by NVIDIA, which allows developers to leverage the power of GPUs for general-purpose computing.
- CUDA programs are written in C/C++ and executed on the GPU.

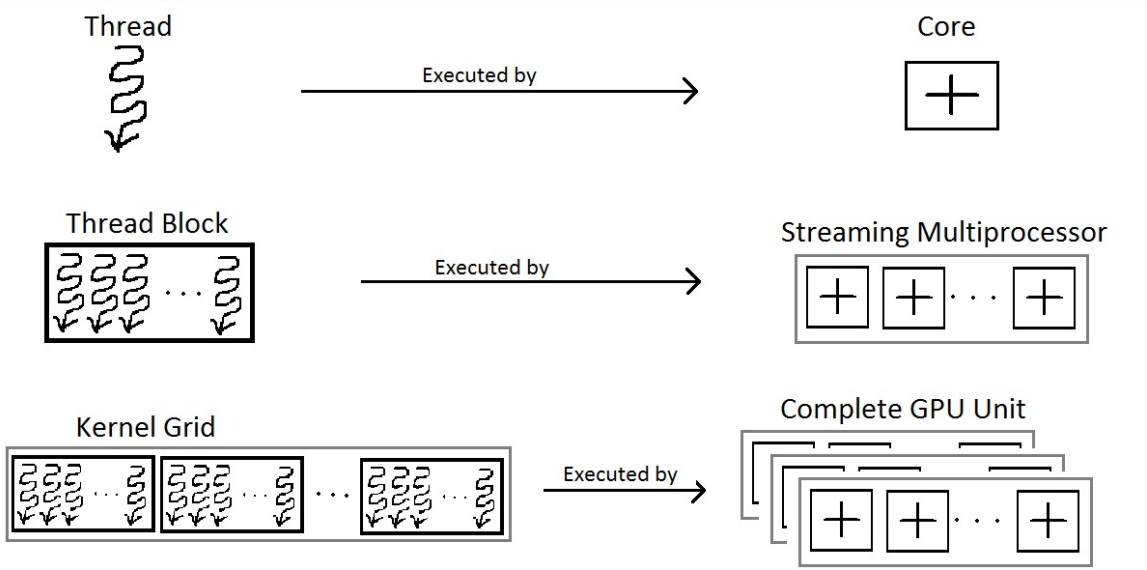
Graphics Processing Units (GPUs)

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



GPU Microarchitecture

- Streaming Multiprocessor (SM): An SM is a processing unit in a GPU that executes multiple threads concurrently. It contains ALUs, registers, and shared memory, and is essential for parallel computations.
- Warp: A warp is a group of threads (typically 32) in a GPU that are executed simultaneously using SIMD. All threads in a warp perform the same instruction on different data elements, maximizing throughput.
- SIMD (Single Instruction, Multiple Data): SIMD is a computing paradigm where one instruction is applied to multiple data elements concurrently. GPUs use SIMD in warps for efficient parallel execution.



CUDA Programming Model

- Kernels
- Threads
- Kernel Launch Parameters

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);
}
```

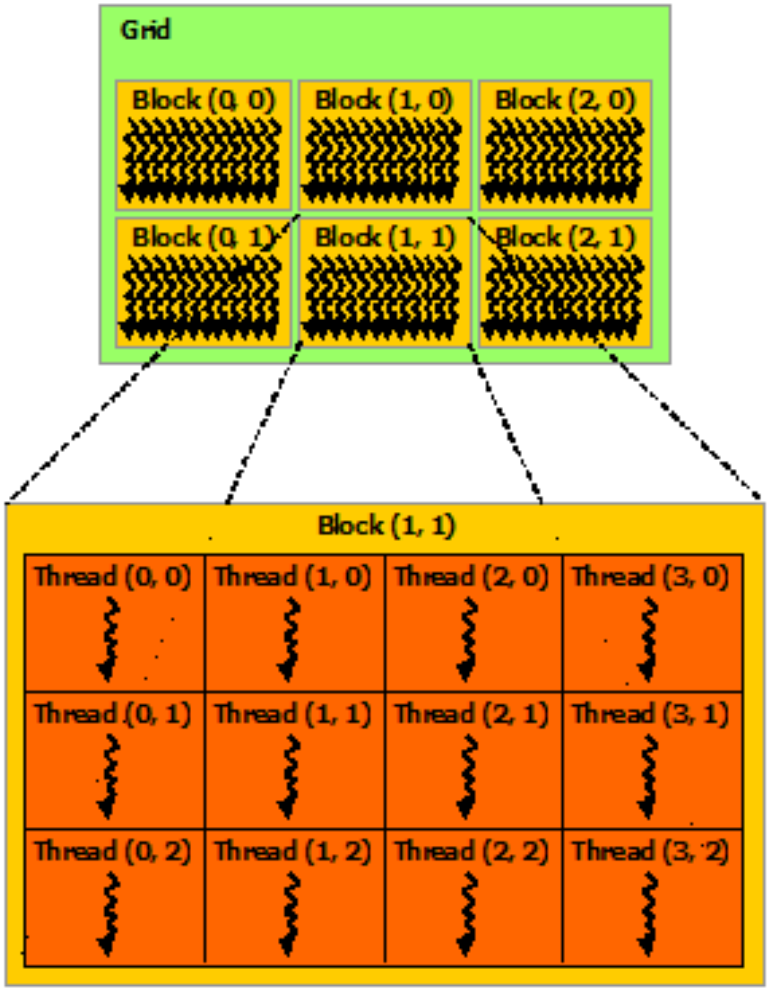
Kernels

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Thread Hiearchy

- Threads constitute the basic units of parallel execution and are organized within a hierarchical structure encompassing threads, thread blocks, and grids.
- Blocks can be executed in any order, both concurrently and sequentially, across the available SMs on a GPU.

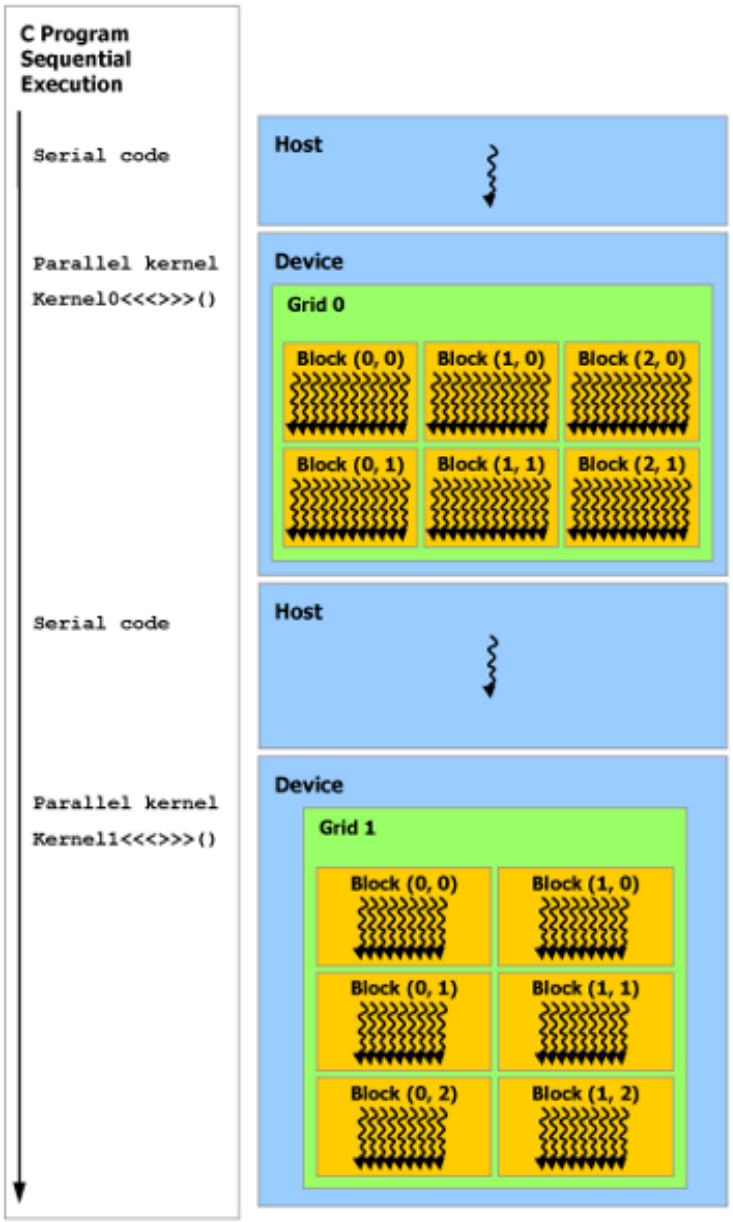


Kernel Launch Parameters

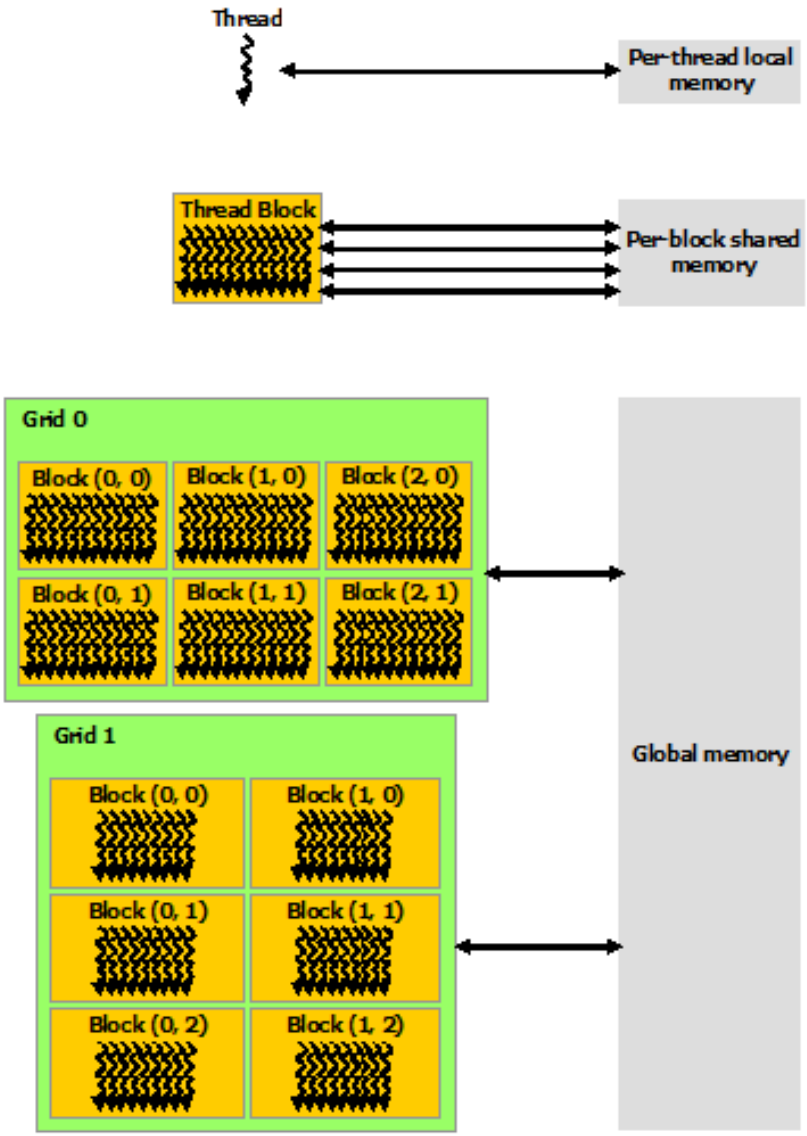
- Kernel launch parameters define the organization of threads and blocks for a particular kernel invocation.
- Optimal kernel launch parameters can significantly improve the performance of a GPU program.
- Finding the best kernel launch parameters is not straightforward, as they depend on the data, hardware, and program characteristics.

```
//a CUDA kernel for vector addition
__global__ void vector_addition(int *A, int *B, size_t n) {
    A[threadIdx.x] += B[threadIdx.x];
}
int main(){
    ...
    //launch the kernel with 1 block and n threads per block
    vector_addition<<<1, n>>>(Ad, Bd, n);
    ...
}
```

Heterogeneous Programming Mode



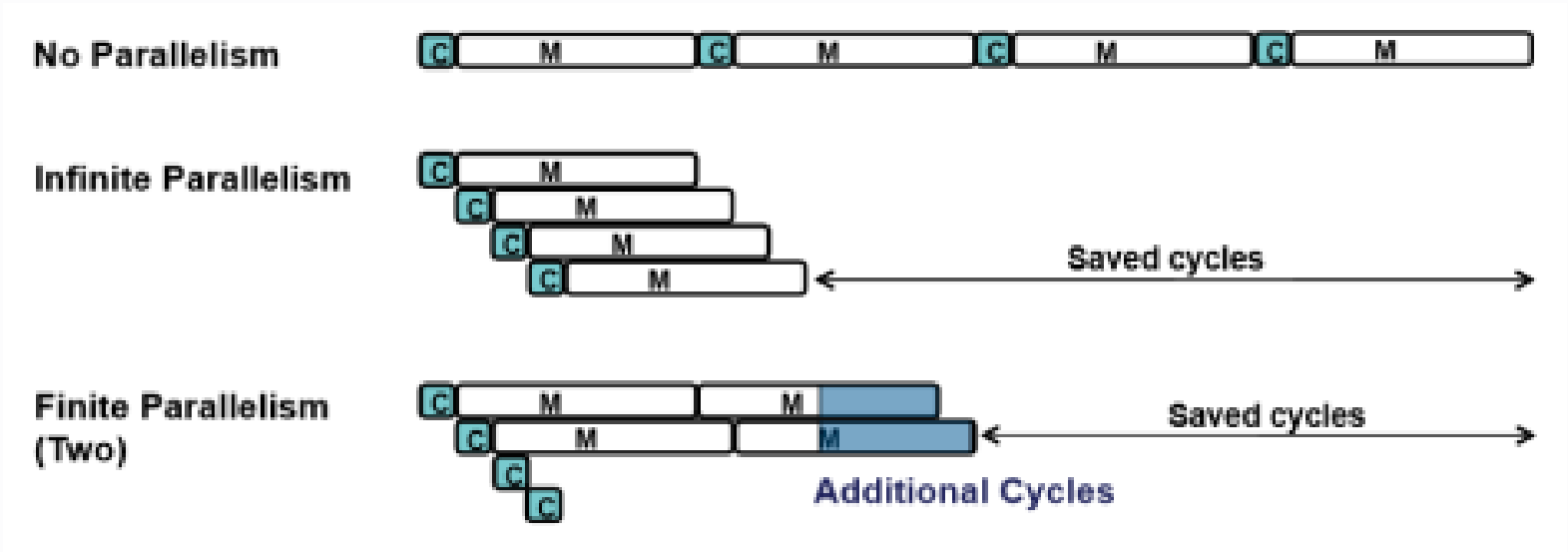
CUDA Memory Model



Outline

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)

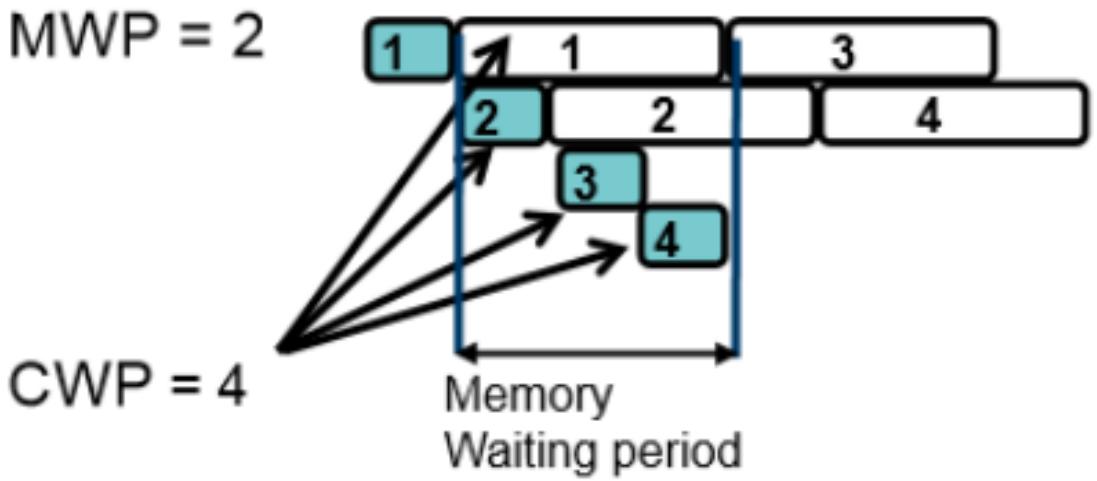
MWP is the maximum number of warps that can overlap memory accesses.

Four warps are overlapped during memory accesses

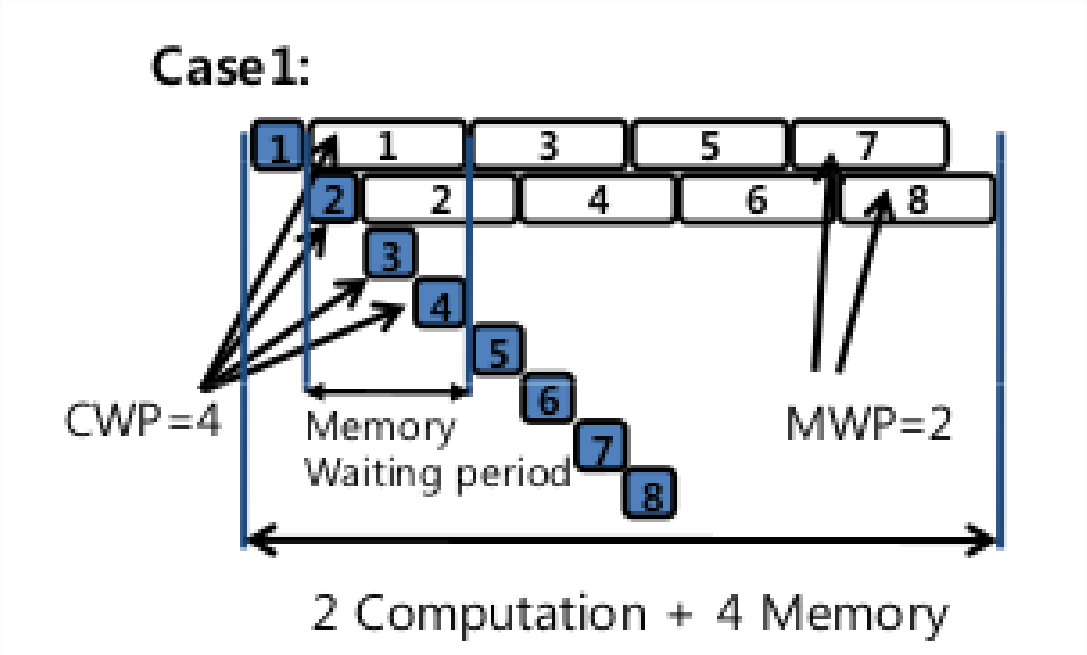
- MWP = 4.
- MWP is determined by #Active SMs, #Active warps, Bandwidth, Types of memory accesses (Coalesced, Uncoalesced)

Computation Warp Parallelism (CWP)

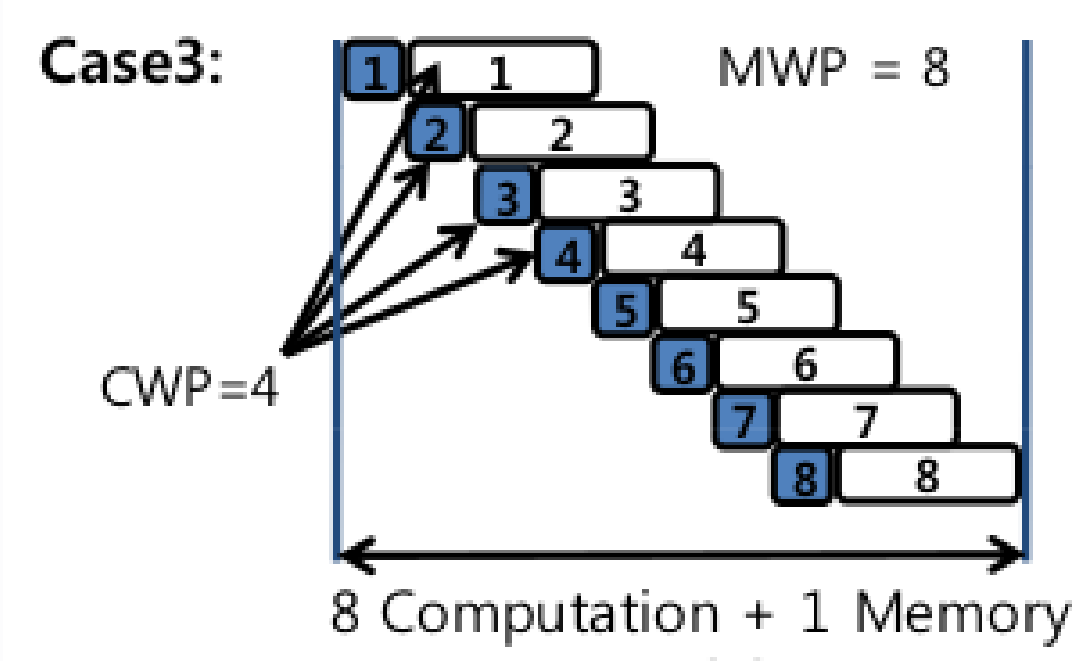
CWP is the number of warps that execute instructions during one memory access period plus one.



- CWP = 4.



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



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

Questions to answer

- For a CUDA program, can we automatically and dynamically finds the values of kernel launch parameters which optimize the performance for each kernel invocation independently?

Outline

Helper vs Rational Program

Note

The terms helper program refers to the generated code which implements the rational prgorams in order to select the optimal thread block configuration for a kernel. Whereas the term rational program refers to the thoery behind modeling a Program.

Rational Programs

- A rational program is a special type of computer program that takes input values $x_1, \ldots, x_n$ and calculates an output value y using a function $f(x_1, \ldots, x_n)$.
- This function is determined by a specific process that distinguishes rational programs from regular ones. In simpler terms, a rational program is a sequence of instructions that work with rational numbers and always produce a rational number as the result.
- To be considered a rational program, the sequence of instructions must follow two conditions:
  (i) The arithmetic operations used in the program must be addition, subtraction, multiplication, division, or comparisons (like equality or ordering) of two rational numbers. These can be done in fixed or arbitrary precision.
  (ii) When you replace the variables $X_1, \ldots, X_n$ with actual rational numbers $x_1, \ldots, x_n$, the program must always come to an end and the final instruction should assign a rational number to the output value Y.

Theoretical Foundations

Parameters influencing the performance of a multithreaded program $\mathcal{P}$

- data parameters($\boldsymbol{D} = (D_1, \ldots, D_d)$), describing size and structure of the data;
- hardware parameters($\boldsymbol{H} = (H_1, \ldots, H_h)$), describing hardware resources and their capabilities; and
- program parameters($\boldsymbol{P} = (P_1, \ldots, P_p)$), characterizing parallel aspects of the program (e.g. how tasks are mapped to hardware resources).

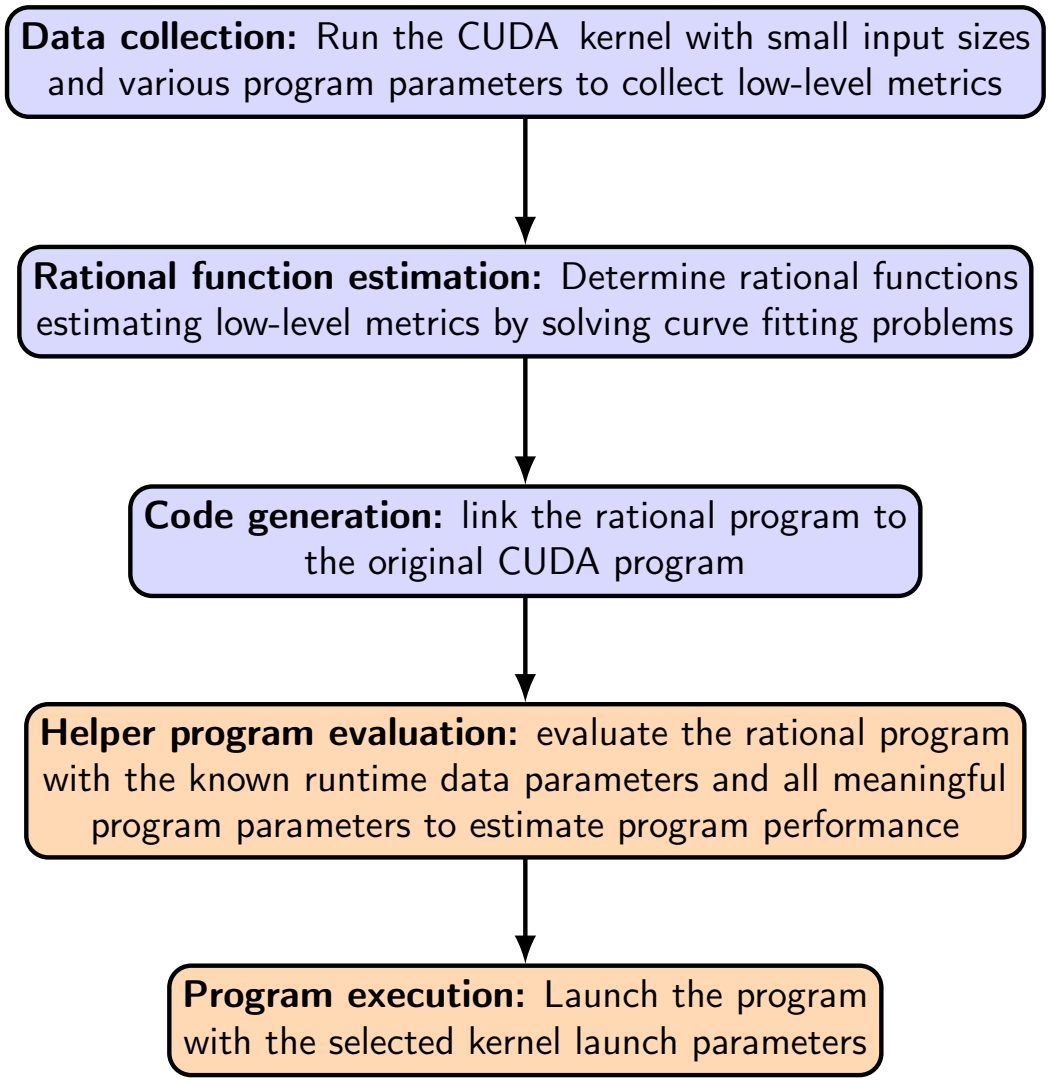
Building a Helper Program

- Let $\mathcal{E}$ be a high-level performance metric (running time, memory consumption)
- g_i can be estimated by curve fitting, if we run the program on some selected $\boldsymbol{D}$ and $\boldsymbol{P}$ and collect the values of L_i
- In order to generate the Helper program $\mathcal{R}$, we proceed as follows:
  - (i) we convert the Helper program representing $\mathcal{E}$ into code,
  - (ii) we convert each $\widehat{g}_i(\boldsymbol{D}, \boldsymbol{P})$ into code, specifically sub-routines, estimating L_i; and
  - (iii) we include those sub-routines into the code computing $\mathcal{E}$, which yields the desired Helper program $\mathcal{R}$, fully constructed, and depending only on $\boldsymbol{D}$ and $\boldsymbol{P}$.

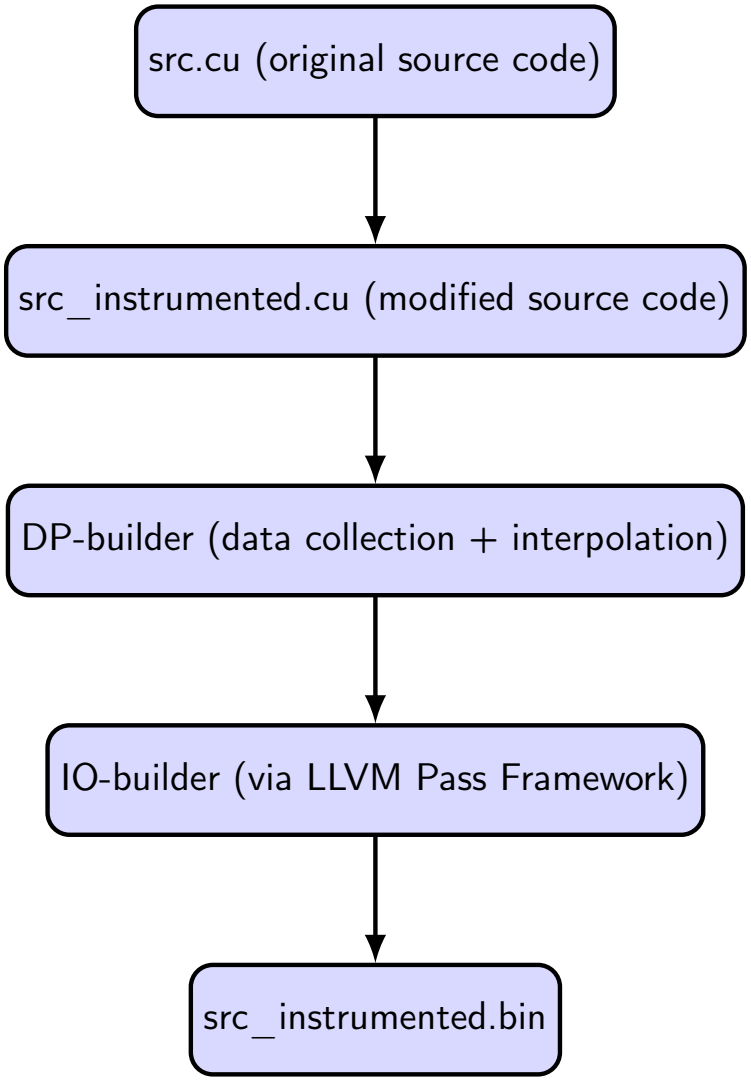
KLARAPTOR

- At the compile-time of a CUDA program, its kernels are analyzed in order to build Helper programs estimating the performance metrics for each individual kernel under the MWP-CWP model. Each helper program, writing as code in C programming language, is inserted into the code of the CUDA program so that it is called before the execution of the corresponding kernel
- At run-time, immediately preceding the launch of a kernel, where D is known, the helper program is evaluated to determine the thread block configuration which optimizes the performance of the kernel. The kernel is then launched using this configuration.

Steps of KLARAPTOR



High-level Design of KLARAPTOR



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.

Outline

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.
- Add two pragmas for each kernel, specifying kernel dimension and the index of the kernel input argument corresponding to the data size N

Input/Output builder

The Input/Output Builder Pass creates an "instrumented binary" that embeds an IO mechanism to communicate with the helper program for each kernel invocation, obtaining optimal kernel launch parameters at runtime.

- (i) Obtain the LLVM intermediate representation (IR) of the instrumented source code and find all CUDA driver API kernel calls.
- (ii) Using the annotated information for each kernel, determine which variables in the IR contain the value of N for a corresponding kernel call.
- (iii) Insert a call to an IO function immediately before each kernel call to pass the runtime value of N to the corresponding helper program and retrieve the optimal kernel launch parameters.

Building a helper program: data collection

Data collection is necessary to perform rational function approximation and gather metrics on performance counters and runtime metrics.

- (i) Architecture-specific performance counters influenced by the Compute Capability (CC) of the device.
- (ii) Runtime-specific performance counters that depend on the kernel's behavior, including memory access patterns and the number of executed warps.
- (iii) Device-specific parameters describing the actual GPU card, including memory bandwidth, departure delay for memory accesses, number of SMs, clock frequency, and instruction delay.

Building a helper program: data collection

Runtime-specific performance counters

```
[trace: n=4096, bx=32, by=8, elapsed_Convolution2D_kernel=219.9013 (ms)] ... PASS
==PROF== Disconnected from process 41150
"Metric Name","Metric Unit","Metric Value"
"dram__bytes_read.sum","byte","68,456,288"
"dram__bytes_write.sum","byte","66,864,032"
"l1tex__t_bytes_pipe_lsu_mem_global_op_ld.sum.per_second","byte/second","1,293,146,285,512.
"l1tex__t_bytes_pipe_lsu_mem_global_op_st.sum.per_second","byte/second","123,294,394,447.39
"l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum","sector","21,984,780"
"l1tex__t_sectors_pipe_lsu_mem_global_op_st.sum","sector","2,096,128"
"launch__block_dim_x","block","32"
"launch__block_dim_y","block","8"
"launch__block_dim_z","block","1"
"launch__grid_dim_x","","128"
"launch__grid_dim_y","","512"
"launch__grid_dim_z","","1"
"launch__registers_per_thread","register/thread","30"
"launch__shared_mem_per_block_dynamic","byte/block","0"
"launch__shared_mem_per_block_static","byte/block","0"
"sm__warps_active.avg.pct_of_peak_sustained_active","%","87.46"
"smsp__average_inst_executed_per_warp.ratio","inst/warp","46.98"
"smsp__inst_executed.avg.per_cycle_active","inst/cycle","0.18"
"smsp__inst_executed_op_global_ld.sum","inst","4,716,288"
"smsp__inst_executed_op_global_st.sum","inst","524,032"
```

Building a helper program: data collection

Device-specific parameters

```
[Issue_cycles: 1]
[Mem_bandwidth: 448.06]
[Mem_LD: 501]
[Departure_del_uncoal: 2]
[Departure_del_coal: 2]
[Active_SMs: 40]
[Freq: 1785]
[Load_bytes_per_warp: 128]

[device_name: NVIDIA GeForce RTX 2070 SUPER]
[driver_version: 11.7]
[runtim_version: 11.7]
[compute_capability: 7.5]
[global_memory_bytes: 8361803776]
[n_cores_per_sm:  64]
[n_blocks_per_sm: 32]
[n_cores: 2560]
[memory_clock_rate_ghz: 7.00]
[memory_bus_width_bits: 256]
[l2_cache_size_bytes: 4194304]
[total_constant memory_bytes: 65536]
[total_shared_memory_per_block_bytes: 49152]
[total_registers_available_per_block: 65536]
[warp_size: 32]
```

Building a helper program: outlier removal

- Outlier removal addresses potential noise in the empirical data gathered from NVIDIA's Nsight Compute (ncu) and enhances the accuracy of the parameter estimation process.
  - (i) Profile the program for small input sizes and obtain MWP-CWP estimation for clock-cycles for various thread block configurations.
  - (ii) Calculate Q1, Q3, and IQR for the estimated clock-cycles and determine the upper inner fence.
  - (iii) Identify and remove outliers exceeding the upper inner fence threshold.

```
128 1 32 72869.600000
128 1 64 78474.600000
128 1 128 89684.600000
256 1 64 294064.600000
256 1 32 361929.200000
512 1 32 1271232.200000
512 1 64 1452818.400000
1024 1 64 4729979.800000
1024 1 32 4730039.600000
2048 1 32 18771193.800000
2048 1 64 18953199.200000
2048 1 512 18953199.200000
2048 1 1024 18953199.200000
```

Building a helper program: rational function approximation

- The goal of this step is to determine rational functions that estimate low-level performance metrics or other intermediate values in the rational program $\mathcal{R}$. These rational functions are used to build the objective function $\mathcal{E} = f(\boldsymbol{P}, \boldsymbol{D}, \boldsymbol{H})$
- A rational function is a fraction of two polynomials, with defined degree bounds on each variable, the polynomials can be represented by coefficients that we aim to estimate:

$$f(X_1, \ldots, X_n) = \frac{\alpha_1 \cdot (X_1^0 \cdots X_n^0) \ + \ \ldots \ + \ \alpha_i \cdot (X_1^{u_1} \cdots X_n^{u_n})}{\beta_1 \cdot (X_1^0 \cdots X_n^0) \ + \ \ldots \ + \ \beta_j \cdot (X_1^{v_1} \cdots X_n^{v_n})} \tag{1}$$

- We perform parameter estimation on the coefficients of the polynomials in the rational function. The estimation process involves solving an over-determined system of linear equations.
- We use numerical analysis techniques such as singular value decomposition (SVD) and linear least squares to solve the system.

Outline

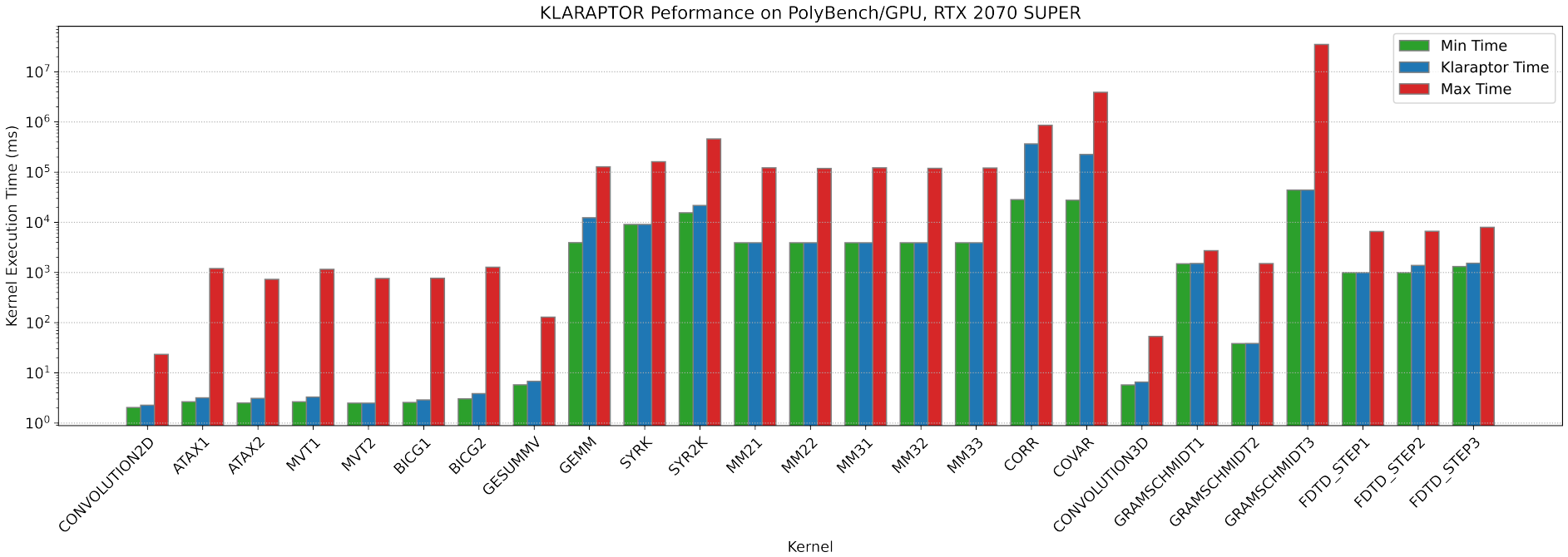


Figure: Comparing kernel execution time (log-scaled) for the thread block configuration chosen by KLARAPTOR versus the minimum and maximum times as determined by an exhaustive search over all possible configurations. Kernels are part of the `PolyBench/GPU` benchmark suite and executed on a RTX 2070 SUPER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

Summary

KLARAPTOR: A Tool for Dynamically Finding Optimal Kernel Launch Parameters Targeting CUDA Programs

- We present KLARAPTOR, a freely available tool built on top of the LLVM Pass Framework and NVIDIA Nsight Comute CLI to dynamically determine the optimal values of kernel launch parameters of a CUDA kernel.
- We describe a technique to build at the compile-time of a CUDA program a so-called helper program. The helper program, based on some performance prediction model, and knowing particular data and hardware parameters at runtime, can be executed to automatically and dynamically determine the values of launch parameters for the CUDA program.
- Our underlying technique could be applied to parallel programs in general, given a performance prediction model which accounts for program and hardware parameters.
- We have implemented and successfully tested our technique in the context of GPU kernels written in CUDA.

Future work

KLARAPTOR

- Investigate the complexity of efficiently utilizing available hardware resources in GPU execution.
- Reevaluate the use of linear least squares for extrapolation beyond the "training" range.
- Test the model with multiples of 32 within the training range to more accurately capture CUDA thread block dimensions.
- Investigate random forest regression as an alternative approach, given the tree-like structure of rational programs.
- Explore the potential of machine learning models, such as neural networks, for determining optimal thread block configurations.

Thank You!

Your Questions?


A. Brandt, D. Mohajerani, M. M. Maza, J. Paudel, and L. Wang.
KLARAPTOR: A tool for dynamically finding optimal kernel launch parameters targeting CUDA programs.
CoRR, abs/1911.02373, 2019.