A Technique for Finding Optimal Program Launch Parameters Targeting Manycore Accelerators

CAS Project Number 880: Comprehensive Optimization of Parametric Kernels for Graphics Processing Units

Product: XL C/C++ for Linux, XL Fortran for Linux

Alexander Brandt, Davood Mohajerani, Marc Moreno Maza, Jeeva Paudel, Linxiao Wang

Introduction

Three types of parameters influence the performance of parallel programs on multiprocessors:

- **data parameters**, such as input data and its size,
- **hardware parameters**, such as cache capacity and number of available registers, and
- **program parameters**, such as granularity of tasks and the quantities that characterize how tasks are mapped to processors.

The choice of program parameters can significantly affect the performance of the program. It is therefore critical to find optimal program parameter values that yield the best performance for a given confluence of hardware and data parameter values. We present a novel technique to statically build a program $\mathcal{R}$ that can dynamically determine the optimal program parameter values to yield the best program performance for given values of the data and hardware parameters of a given multithreaded program $\mathcal{P}$.

To be precise, let $\mathcal{E}$ be a performance metric for $\mathcal{P}$ we want to optimize. Given data parameters values $D_1, \ldots, D_d$, the goal is to find program parameter values $P_1, \ldots, P_p$ such that the execution of $\mathcal{P}$ optimizes $\mathcal{E}$. To accomplish this, we compute a mathematical expression, parameterized by data and program parameters, in the format of a **rational program** $\mathcal{R}$ (computed at compile-time). At runtime, given specific values of $D_1, \ldots, D_d$, we can efficiently evaluate $\mathcal{E}$ using $\mathcal{R}$. We can then feasibly determine values of $P_1, \ldots, P_p$ optimizing $\mathcal{E}$, and feed them to $\mathcal{P}$ for execution.

Rational Program

Let $X_1, \ldots, X_n, Y$ be pairwise different variables. Let $\mathcal{S}$ be a sequence of three-address code instructions such that the set of the variables that occur in $\mathcal{S}$ and are never assigned a value by an instruction of $\mathcal{S}$ is exactly $\{X_1, \ldots, X_n\}$.

Definition We say that the sequence $\mathcal{S}$ is rational if every arithmetic operation used in $\mathcal{S}$ is either an addition, a subtraction, a multiplication, or a comparison ($=$, $<$), for integer numbers either in fixed precision or in arbitrary precision. Moreover, we say that the sequence $\mathcal{S}$ is a rational program in $X_1, \ldots, X_n$ evaluating Y if the following two conditions hold:

(i) $\mathcal{S}$ is rational, and

(ii) after specializing each of $X_1, \ldots, X_n$ to an arbitrary integer value in $\mathcal{S}$, the execution of the specialized sequence $\mathcal{S}$ always terminates and the last executed instruction assigns an integer value to Y.

One can easily extend the above definition by allowing the use of the Euclidean division for integers. Also the definition can be extended for rational numbers.

Example

Hardware occupancy, as defined in CUDA, is a measure of how effectively a program is making use of the hardware's processors (Streaming Multiprocessors in case of GPUs). Occupancy is calculated from the hardware parameters:

- the maximum number $R_{\max}$ of registers per thread block,
- the maximum number $Z_{\max}$ of shared memory words per thread block,
- the maximum number $T_{\max}$ of threads per thread block,
- the maximum number $B_{\max}$ of thread blocks per Streaming Multiprocessor (SM) and
- the maximum number $W_{\max}$ of warps per SM,

as well as low-level performance metrics, namely:

- the number R of registers used per thread and
- the number Z of shared memory used per thread block,

and a program parameter, namely the number T of threads per thread block. The hardware occupancy of a CUDA kernel is defined as the ratio between the number of active warps per SM and the maximum number of warps per SM, namely $W_{\text{active}}/W_{\max}$, where

$$W_{\text{active}} = \min\left(\lfloor B_{\text{active}} T/32 \rfloor, W_{\max}\right)$$

and B_{active} is given as a flow chart by Figure 1.

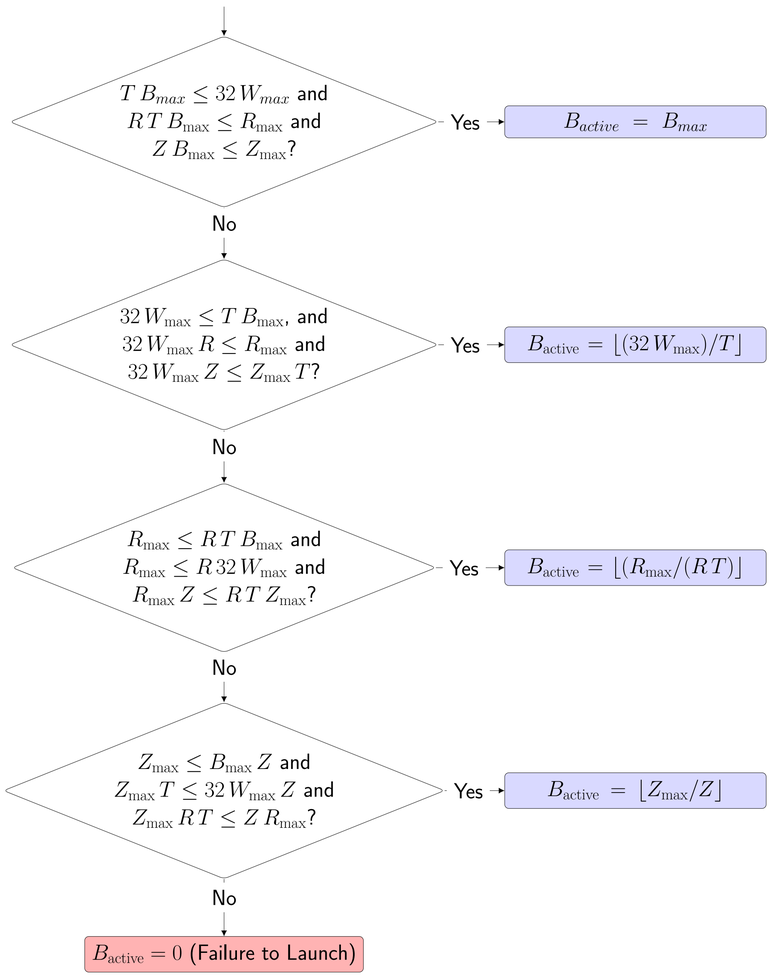


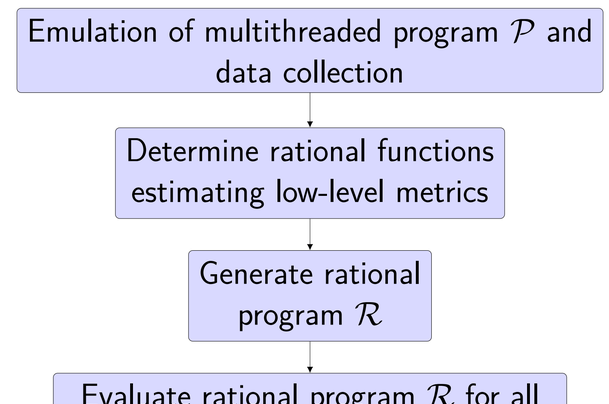
Figure: Rational program (presented as a flow chart) for the calculation of hardware occupancy in CUDA.

Algorithm

We show a step-by-step description of our proposed approach for building a rational program $\mathcal{R}$. We assume that:

- $\mathcal{E}$ is a high-level performance metric for the multithreaded program $\mathcal{P}$ (e.g. execution time, memory consumption, and hardware occupancy),
- $\mathcal{E}$ is given (by a program execution model, e.g. MWP-CWP) as a rational program depending on hardware parameters $H_1, \ldots, H_h$, low-level performance metrics $L_1, \ldots, L_\ell$, and program parameters $P_1, \ldots, P_p$,
- the values of the hardware parameters are known at compile-time for $\mathcal{P}$ while the values of the data parameters $D_1, \ldots, D_d$ are known at runtime for $\mathcal{P}$,
- the data and program parameters take integer values.

The entire process is decomposed below into **six** steps: the first three take place at compile-time while the other three are performed at execution-time.



1. **Data collection**: We select a set of points $K_1, \ldots, K_k$ in the space of the possible values of $(D_1, \ldots, D_d, P_1, \ldots, P_p)$ We call this space F. Then we run (or emulate) the program $\mathcal{P}$ on these points and measure the low-level performance metrics $L_1, \ldots, L_\ell$; and for each $1 \le i \le \ell$, we record the values $(v_{i,1}, \ldots, v_{i,k})$ measured for L_i at the respective points $K_1, \ldots, K_k$.
2. **Rational function determination**: For each $1 \le i \le \ell$, we use the values $(v_{i,1}, \ldots, v_{i,k})$ measured for L_i at the respective points $K_1, \ldots, K_k$ to estimate the rational function $g_i(D_1, \ldots, D_d, P_1, \ldots, P_p)$ using curve fitting (e.g. by linear least squares).

Experimentation

We implemented our idea for CUDA kernels using the MWP-CWP model and NVIDIA's `nvprof` for data collection. Tests use the `PolyBench` benchmarking suite.

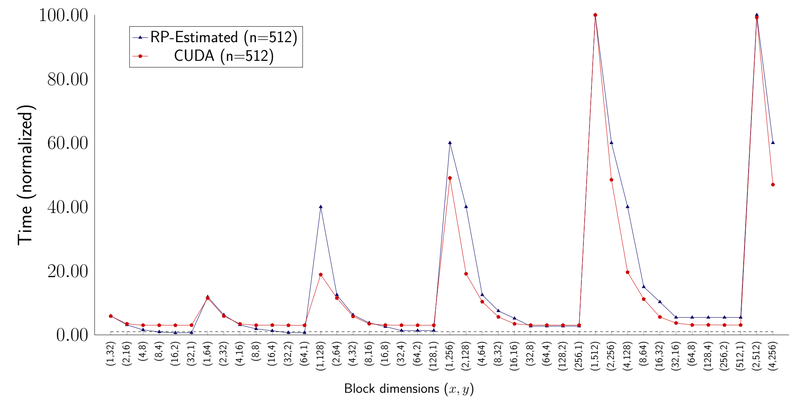


Figure: Example of Kernel atax_kernel2 from Polybench suite.

ID	Data param	Default Config		Rational Program		best	worst	Error (%)
		prog param	time	prog param	time			
1	1024	32×8	0.04	32×8	0.04	0.04	0.34	0.00
	2048	32×8	0.16	128×4	0.16	0.15	1.31	0.86
2.1	1024	32×8	18.2	32×1	23.4	18.1	84.0	7.93
2.2	1024	32×8	19.9	64×2	19.7	19.5	82.3	0.26
2.3	1024	32×8	25.7	16×2	27.2	25.7	109	1.79
3	1024	32×8	5.73	16×2	9.67	5.73	89.0	4.73
	2048	32×8	46.7	16×2	78.6	46.7	707	4.83
4	1024	32×8	5.77	16×2	9.68	5.73	89.8	4.69
	2048	32×8	47.3	16×2	78.8	47.1	731	4.63
5.1	1024	256×1	0.25	32×1	0.25	0.25	9.27	0.01
	2048	256×1	0.51	32×1	0.50	0.50	37.2	0.00
5.2	1024	256×1	0.19	32×1	0.18	0.16	9.48	0.17
	2048	256×1	0.38	32×1	0.36	0.36	38.5	0.01
6	1024	32×8	5.76	32×1	9.69	5.76	95.6	4.37
	2048	32×8	46.3	32×1	78.3	46.3	725	4.70
7	1024	32×8	56.0	8×4	81.6	55.2	369	8.38
8.1	1024	256×1	0.20	32×1	0.19	0.17	9.87	0.15
	2048	256×1	0.39	32×1	0.37	0.37	42.5	0.00
8.2	1024	256×1	0.26	32×1	0.24	0.24	9.35	0.03
	2048	256×1	0.50	32×1	0.49	0.49	40.3	0.01
9	1024	256×1	0.40	32×1	0.37	0.34	2.10	1.53
	2048	256×1	0.80	32×1	0.74	0.73	6.32	0.08
10	1024	32×8	0.02	32×1	34.3	17.9	87.9	23.38
	2048	32×8	239	32×1	968	143	2802	31.03
11.1	1024	256×1	0.20	32×1	0.19	0.17	9.83	0.17
	2048	256×1	0.39	32×1	0.38	0.36	39.3	0.03
11.2	1024	256×1	0.25	32×1	0.24	0.24	9.55	0.02
	2048	256×1	0.50	32×1	0.49	0.49	37.4	0.00
12	1024	32×8	90.5	32×1	338	32.0	338	100.0
	2048	32×8	2295	32×1	5276	251	6723	77.64
13.1	1024	256×1	320	8×8	256	224	2461	1.42
	2048	256×1	1359	1×128	7410	1186	18361	36.23
13.2	1024	256×1	0.26	32×1	0.26	0.25	1.98	0.17
	2048	256×1	0.52	32×1	0.51	0.51	7.22	0.00
13.3	1024	32×8	0.05	1×128	0.02	0.02	0.21	0.00
	2048	32×8	0.21	1×128	0.10	0.10	0.85	0.13
13.4	1024	256×1	0.26	32×1	0.25	0.25	2.18	0.15
	2048	256×1	0.53	32×1	0.52	0.52	8.16	0.00
14.1	1024	256×1	1065	32×8	394	226	2462	7.54
	2048	256×1	2116	64×16	6119	1182	18343	28.76
14.2	1024	256×1	1.63	64×1	0.26	0.25	2.30	0.39
	2048	256×1	3.29	32×1	0.51	0.51	7.58	0.05
14.3	1024	32×8	0.03	128×1	0.01	0.01	1.48	0.00
	2048	32×8	0.11	128×1	0.01	0.01	5.94	0.00
15.1	1024	256×1	17.3	512×1	17.5	17.1	23.1	7.27
	2048	256×1	82.8	512×1	83.1	82.7	87.3	7.98
15.2	1024	256×1	3.03	64×1	2.93	2.93	4.34	0.00
	2048	256×1	6.53	32×1	6.64	6.36	11.5	5.47
15.3	1024	256×1	389	32×1	375	375	522	0.00
	2048	256×1	2000	32×1	1947	1944	3818	0.11

Table: Comparing default thread block configurations to those computed by the rational program.

Algorithm (Cont.)

3. **Code generation**: Each rational function is converted into sub-routines for evaluating one of $L_1, \ldots, L_\ell$. Those sub-routines are included into code for computing $\mathcal{E}$ based on some model, yielding the desired rational program $\mathcal{R}$.
4. **Rational program evaluation**: At execution time the data parameters $D_1, \ldots, D_d$ now have known, specific values, say $\delta_1, \ldots, \delta_d$. With those data parameter values and possible program parameter values the rational program $\mathcal{R}$ can be run to evaluate $\mathcal{E}$.
5. **Selection of optimal values of program parameters**: Using either exhaustive search or a numerical optimization technique we can determine values of the program parameters which optimize $\mathcal{E}$.
6. **Program execution**: Executing $\mathcal{P}$ using the selected configuration of program parameters $P_1, \ldots, P_p$ along with with the values $\delta_1, \ldots, \delta_d$ of $D_1, \ldots, D_d$.

Acknowledgements

This work is supported by IBM Canada Ltd (CAS project 880) and NSERC of Canada (CRD grant CRDPJ500717-16).