Università
della
Svizzera
italiana

Faculty
of Informatics

USI Technical Report Series in Informatics

Adapting `cuda_launch_config` compute grid configurator for multidimensional plain and compiler-generated GPU kernels

Zdenek Pesek[1,2]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland
[2] Faculty of Information Technology, Czech Technical University in Prague

Abstract

Exploiting the potential of GPU computing is inevitable for any modern HPC application. One of the crucial aspects of programming efficient CUDA kernels is choosing the optimal compute grid/block configuration. Hard-coded configuration of block size (e.g. {128, 1, 1}) is known to give good performance results on Fermi and Kepler GPUs, however faster configuration could potentially be found for each individual kernel, if its specific properties are accounted.

Our work is based on `cuda_launch_config` – an existing framework for heuristic search of semi-optimal grid/block configuration, originally designed for 1-dimensional Thrust kernels. We modified it to determine optimal setups for multidimensional kernels and added an interface layer to use it inside KernelGen GPU auto-parallelizing compiler. These contributions allowed us to improve the performance of GPU kernels of stencil benchmark by up to XX% in case of manually-written CUDA code and by up to 7% for kernels generated by KerenGen compiler.

Report Info

Published
July 2014

Number
USI-INF-TR-2014-2

Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

Possible points to cover in introduction:

- The problem of compute grid selection is a major obstacle in understanding CUDA by n00bs
- The concept of occupancy is often discussed in overcomplicated way
- Paper structure
- `cuda_launch_config` is designed for Thrust, and this uses only 1D blocks.

The paper is organized as follows. Section 2 explains the key factors of optimal CUDA compute grid choice.

2 Choosing efficient compute grid

Possible points to cover in this section:

- Key CUDA kernel *code* characteristics: regcount, shared memory

- How regcount and shared memory could become kernel performance limiters (examples)
- What else could be kernel performance limiters
- How to account all limitations together
- cuda_launch_config's approach

3 Implementation for autoparallelizing compiler runtime

- Making cuda_launch_config to play together with runtime constants substitution

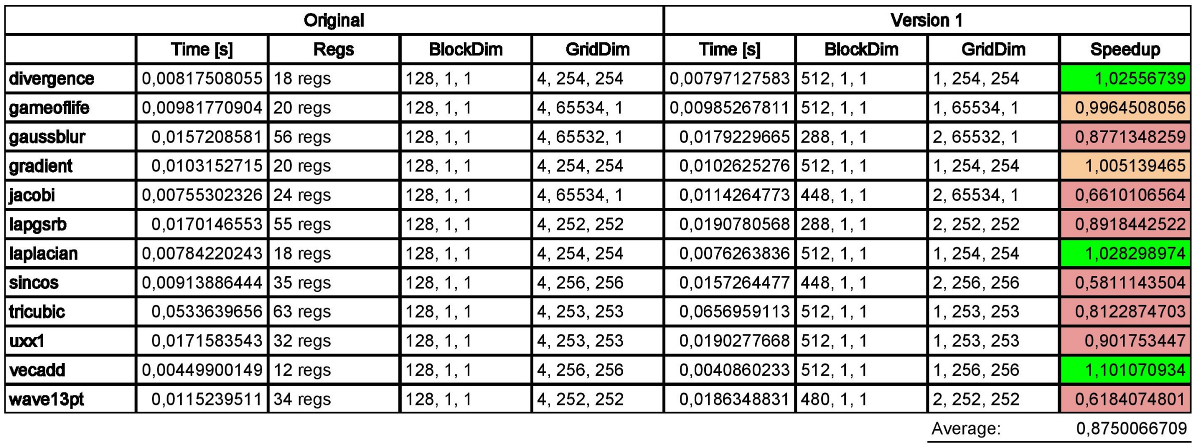
4 Benchmarking

For benchmarking was used the integrated set of tests in KernelGen, which was executed on the Tesla-CMC cluster with NVIDIA Tesla C2075 graphic cards. Execution of the performance tests was handled by the KernelGen script called Buran.

Because of the KernelGen limitations, it was not possible to execute all the need CUDA functions in the cuda_launch_config library. Thus, required parameters had to be supplied externally from the caller function in KernelGen. Almost all these attributes were already present in the code, so they were just passed to the library. The only exception was a maximal threads per block value, which had to be determined heuristically. Consecutive benchmarking showed us, that selection of this value is crucial for the whole algorithm and performance respectively.

4.1 First version

For the first version, the value of threads per block was computed as a closest power of two. When looking at the result chart 1 , it was not the best idea and definitely some more tuning needed to be done. There is a speedup of a 10% in one of the test cases, but the overall result is worse than the origanl version and this is not acceptable if we want to use this version in the real world. However, this initial step gives us a good entry point to start with.



	Original				Version 1			
	Time [s]	Regs	BlockDim	GridDim	Time [s]	BlockDim	GridDim	Speedup
divergence	0,00817508055	18 regs	128, 1, 1	4, 254, 254	0,00797127583	512, 1, 1	1, 254, 254	1,02556739
gameoflife	0,00981770904	20 regs	128, 1, 1	4, 65534, 1	0,00985267811	512, 1, 1	1, 65534, 1	0,9964508056
gaussblur	0,0157208581	56 regs	128, 1, 1	4, 65532, 1	0,0179229665	288, 1, 1	2, 65532, 1	0,8771348259
gradient	0,0103152715	20 regs	128, 1, 1	4, 254, 254	0,0102625276	512, 1, 1	1, 254, 254	1,005139465
jacobi	0,00755302326	24 regs	128, 1, 1	4, 65534, 1	0,0114264773	448, 1, 1	2, 65534, 1	0,6610106564
lapgsrb	0,0170146553	55 regs	128, 1, 1	4, 252, 252	0,0190780568	288, 1, 1	2, 252, 252	0,8918442522
laplacian	0,00784220243	18 regs	128, 1, 1	4, 254, 254	0,0076263836	512, 1, 1	1, 254, 254	1,028298974
sincos	0,00913886444	35 regs	128, 1, 1	4, 256, 256	0,0157264477	448, 1, 1	2, 256, 256	0,5811143504
tricubic	0,0533639656	63 regs	128, 1, 1	4, 253, 253	0,0656959113	512, 1, 1	1, 253, 253	0,8122874703
uxx1	0,0171583543	32 regs	128, 1, 1	4, 253, 253	0,0190277668	512, 1, 1	1, 253, 253	0,901753447
vecadd	0,00449900149	12 regs	128, 1, 1	4, 256, 256	0,0040860233	512, 1, 1	1, 256, 256	1,101070934
wave13pt	0,0115239511	34 regs	128, 1, 1	4, 252, 252	0,0186348831	480, 1, 1	2, 252, 252	0,6184074801
							Average:	0,8750066709

Figure 1: Comparsion of the original KernelGen results and the first version with cuda_launch_config

4.2 Second version

The second version showed 2 the impact, a correct selection of a block grid can have. The limiting factor was now calculated as a nearest power of two lower than the the input value. There are signifacnt differences in speedup when compared to the first version. But, deviations from the normal can be seen in both ways. Especially in the sincos benchmark, the result is very bad. Assuming, that the original hard-coded value 128 was a good estimation of an average value, the resulting block grid is still to big. The overall performance is now almost equal to the original version, but we want to see there a number clearly higher than one. Also, results of the test cases should be more consistent - ideally same or better than the original version for majority of test cases.

	Original				Version 2			
	Time [s]	Regs	BlockDim	GridDim	Time [s]	BlockDim	GridDim	Speedup
divergence	0,00817508055	18 regs	128, 1, 1	4, 254, 254	0,00711087785	256, 1, 1	2, 254, 254	1,149658414
gameoflife	0,00981770904	20 regs	128, 1, 1	4, 65534, 1	0,00928980966	256, 1, 1	2, 65534, 1	1,05682564
gaussblur	0,0157208581	56 regs	128, 1, 1	4, 65532, 1	0,0173936397	192, 1, 1	3, 65532, 1	0,9038279722
gradient	0,0103152715	20 regs	128, 1, 1	4, 254, 254	0,00833435578	256, 1, 1	2, 254, 254	1,237680724
jacobi	0,00755302326	24 regs	128, 1, 1	4, 65534, 1	0,00888222508	224, 1, 1	3, 65534, 1	0,8503526078
lapgsrb	0,0170146553	55 regs	128, 1, 1	4, 252, 252	0,0182919665	192, 1, 1	3, 252, 252	0,9301709196
laplacian	0,00784220243	18 regs	128, 1, 1	4, 254, 254	0,00696327087	256, 1, 1	2, 254, 254	1,126223951
sincos	0,00913886444	35 regs	128, 1, 1	4, 256, 256	0,0157162371	224, 1, 1	3, 256, 256	0,5814918916
tricubic	0,0533639656	63 regs	128, 1, 1	4, 253, 253	0,057009351	256, 1, 1	2, 253, 253	0,9360563603
uxx1	0,0171583543	32 regs	128, 1, 1	4, 253, 253	0,016897139	256, 1, 1	2, 253, 253	1,015459144
vecadd	0,00449900149	12 regs	128, 1, 1	4, 256, 256	0,00348925417	256, 1, 1	2, 256, 256	1,289387723
wave13pt	0,0115239511	34 regs	128, 1, 1	4, 252, 252	0,0120100202	192, 1, 1	3, 252, 252	0,9595280364
							Average:	1,003055282

Figure 2: Comparsion of the original KernelGen results and the second version with cuda_launch_config

4.3 Third version

The third and final version performed really well 3. It produced the best results, which could be suitable for the production version use. Requirements stated when commenting the previous chart were met in this case. Speedup values are now more consistent. Appart from a single performance test, all generated block configurations are resulting in the same or even better performance than in the original version. A key to this final step was lowering the maximal threads per threads per block value by one additional shift right operation. For the first time, we can see even lower number of threads in the block grid than the original 128. Both tests with the block grid set to 96 are significantly faster than before. The final average improvement is almost 7%.

	Original				Version 3			
	Time [s]	Regs	BlockDim	GridDim	Time [s]	BlockDim	GridDim	Speedup
divergence	0,00817508055	18 regs	128, 1, 1	4, 254, 254	0,00815069917	128, 1, 1	4, 254, 254	1,002991324
gameoflife	0,00981770904	20 regs	128, 1, 1	4, 65534, 1	0,00981872655	128, 1, 1	4, 65534, 1	0,9998963705
gaussblur	0,0157208581	56 regs	128, 1, 1	4, 65532, 1	0,0114586219	96, 1, 1	6, 65532, 1	1,37196761
gradient	0,0103152715	20 regs	128, 1, 1	4, 254, 254	0,0103134828	128, 1, 1	4, 254, 254	1,000173433
jacobi	0,00755302326	24 regs	128, 1, 1	4, 65534, 1	0,0075341991	128, 1, 1	4, 65534, 1	1,002498495
lapgsrb	0,0170146553	55 regs	128, 1, 1	4, 252, 252	0,0127252364	96, 1, 1	6, 252, 252	1,337079703
laplacian	0,00784220243	18 regs	128, 1, 1	4, 254, 254	0,00780907118	128, 1, 1	4, 254, 254	1,004242662
sincos	0,00913886444	35 regs	128, 1, 1	4, 256, 256	0,0114679052	224, 1, 1	3, 256, 256	0,7969079165
tricubic	0,0533639656	63 regs	128, 1, 1	4, 253, 253	0,053449442	128, 1, 1	4, 253, 253	0,998400799
uxx1	0,0171583543	32 regs	128, 1, 1	4, 253, 253	0,0171085195	128, 1, 1	4, 253, 253	1,002912865
vecadd	0,00449900149	12 regs	128, 1, 1	4, 256, 256	0,00348766659	256, 1, 1	2, 256, 256	1,28997465
wave13pt	0,0115239511	34 regs	128, 1, 1	4, 252, 252	0,011500105	128, 1, 1	4, 252, 252	1,002073555
							Average:	1,067426615

Figure 3: Comparsion of the original KernelGen results and the third version with cuda_launch_config

5 Conclusion

To summarize this report we need to look back at the beginning, where the important question was stated - Can be cuda_launch_config project library successfully used for a large-scale kernel configuration generation? According to the benchmarking results, automatic block grid generation was the right step forward. It also proved, that the occupancy metric was selected correctly to evaluate theoretical kernel performance and has a great impact on the final result. Two of the three speedups were achieved with a block size of 96 threads, which is even lower value than the original 128 threads. These two kernels are using a lot of registers on the GPU, which are strictly limited. This means, that only few kernels can be assigned to one SM at the same time. Thus, in order to achieve higher occupancy and to be able to assign more kernels to a single SM, we have to decrease the number of threads per block. Average speedup improvement of almost 7% is a very promising result and together with this paper represents a solid base for the possible future work. It would be interesting to investigate the iterative approach used for grid generation.

Acknowledgements (if any) ...

6 Related work

- CUDA occupancy calculator

Google for topic and add related studies, if there are any.