GPURoofline: A Model for Guiding Performance Optimizations on GPUs

Haipeng Jia[1,2], Yunquan Zhang[1,3], Guoping Long[1], Jianliang Xu[2], Shengen Yan[1,3,4], and Yan Li[1,3,4]

[1] Lab. of Parallel Software and Computational Science,Institute of Software, Chinese Academy of Sciences
[2] College of Information Science and Engineering, The Ocean University of China
[3] State Key Laboratory of Computing Science, The Chinese Academy of Sciences
[4] Graduate University of Chinese Academy of Sciences
jiahaipeng95@gmail.com, zyq@mail.rdcps.ac.cn, guoping@iscas.ac.cn

Abstract. Performance optimization on GPUs requires deep technical knowledge of the underlying hardware. Modern GPU architectures are becoming more and more diversified, which further exacerbates the already difficult problem. This paper presents GPURoofline, an empirical model for guiding optimizations on GPUs. The goal is to help non-expert programmers with limited knowledge of GPU architectures implement high performance GPU kernels. The model addresses this problem by exploring potential performance bottlenecks and evaluating whether specific optimization techniques bring any performance improvement. To demonstrate the usage of the model, we optimize four representative kernels with different computation densities, namely matrix transpose, Laplace transform, integral and face-dection, on both NVIDIA and AMD GPUs. Experimental results show that under the guidance of GPURoofline, performance of those kernels achieves 3.74∼14.8 times speedup compared to their naïve implementations on both NVIDIA and AMD GPU platforms.

Keywords: GPURoofline, Threshold Carving, Tradeoff Carving, Little's Law.

1 Introduction

More and more application developers have been adopting GPUs as standard computing accelerators because of their increasing computing power and programmability. However, we won't get the required performance without careful optimizations because the performance problem has shifted from hardware designers to compiler writers and application developers. Unfortunately, performance optimizations of GPU programs are difficult, because this process requires deep technical knowledge of the underlying hardware architecture. Modern GPU architectures are becoming more and more diversified, which further exacerbates the already difficult problem of performance optimization. For programmers, it

© Springer-Verlag Berlin Heidelberg 2012

will be helpful to have a structured and insightful model that guides performance optimizations on GPUs. To make the model even more useful, it needs to be understandable by most programmers.

Our research addresses this problem by proposing GPURoofline, a model guiding performance optimizations on GPUs. The goal of the model is to facilitate the best match between algorithmic features and underlying hardware characteristics. On both NVIDIA and AMD GPUs, the model can help identify performance bottlenecks, and evaluate whether a particular optimization technique can achieve performance improvements. Instead of trying to predict performance, we choose a simpler approach called "bound and bottleneck analysis". The approach provides valuable insights into primary factors affecting the performance. In particular, critical performance bottlenecks are highlighted and quantified [12]. The proposed model provides three functionalities. Firstly it provides valuable insights on primary factors that affect the performance. Secondly it identifies performance bottlenecks and allows programmers and architectures to predict the benefits of potential optimizations and architecture improvements. Thirdly it can be incorporated into a tool to provide performance information to an auto-tuning compiler by narrowing the search space.

We also demonstrate the usage of our model through optimizing four representative programs with different compute intensity: Matrix Transpose, Laplace Transform, Integral and FaceDection. All evaluations are performed on both NVIDIA and AMD GPUs. Experimental results demonstrate that under the guidance of GPURoofline, performance of those kernels achieves 3.74$\sim$14.8 times speedup compared to their naïve implementations on both platforms.

In summary, we make the following contributions in this paper. Firstly, We build the first Roofline model for GPU, called GPURoofline, to guild GPU program optimization. Secondly, We demonstrate how the model can help programmers do GPU performance optimizations. Thirdly,to the best of our knowledge,this is the first performance model that takes global memory channel conflicts and load balancing into consideration.

The rest of the paper is organized as follows. We begin by discussing related works in section 2. Section 3 presents how to build our GPU model. Section 4 discusses experiment results and analysis. Section 5 concludes this paper.

2 Related Work

Enormous works have been invested on building GPU performance analysis and prediction models. Architecture-aware performance analysis methods were proposed in[3][7]. Ryoo et al. [5]used Pareto-optimal curves to narraw the optimization space of GPU programs and introduce efficiency and utilization as single number metrics. N. K. Govindaraju[10]presented a memory model to analyze and improve the performance of nested loops on GPUs. S. Hong[6]presents a simple performance analytical model to capture a rough estimate of the cost of memory operations by considering the number of running threads and memory bandwidth. Baghsorkhi[2]introduced an abstract interpretation of a GPU kernel to identify performance bottlenecks and used work flow graph to predict execution time. Kothapalli[4]presented a performance prediction model to analyze pseudo code for a GPU kernel to obtain a performance estimate. However, because of the complexity of the underlying hardware architecture, it is difficult to predict performance accurately.

Certainly, these performance models are powerful tools for optimizing. However, for a given kernel, they do not provide any insight into how to identify performance bottleneck and evaluate the benefits of potential optimization methods. Compared to them, our work can guide programmers to write high performance program directly, rather than write a naïve version first and then tune it again and again. There are also similar works to us: Yao Zhang[1]provided a quantitative way to analyzes GPU program performance, however, they didn't provide an easy-to-understand model; Samuel Williams[11]provided an insightful visual performance model, however, their works only for multi-core CPUs.

3 GPURoofline

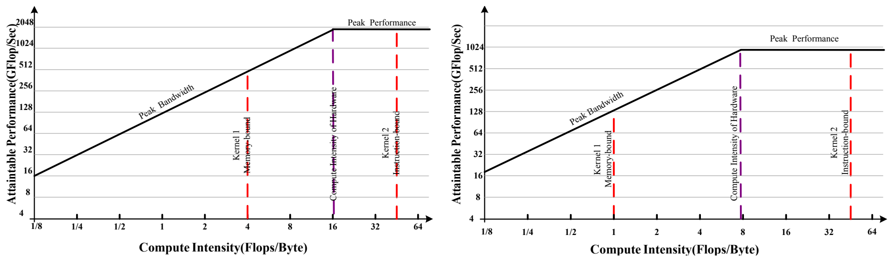
Using bound and bottleneck analysis [8], the attainable performance on a given GPU architecture is restricted by two factors: peak performance and peak bandwidth. Performance depends on how well kernel features map to architectural characteristics. There is a single variable, Compute Intensity, which is defined as operations per byte of off-chip memory traffic. So the proposed GPURoofline model should integrate these three factors together. In this paper, although our work focuses on the NVIDIA Tesla C2050 and AMD Radeon HD5850 GPU, we believe our performance modeling methodology is also applicable to any other GPU architectures.

For simplicity, in this paper, we use peak performance refers to the peak performance of single-precision floating-point, peak bandwidth refers to the peak bandwidth of off-chip memory, NVIDIA GPU refers to the NVIDIA Tesla C2050 GPU and AMD GPU refers to the AMD Radeon HD5850 GPU.

3.1 Naïve GPURoofline

Fig.1a outlines a naïve GpuRoofline model for AMD GPU with peak performance of 2.09TFlopps/sec and peak bandwidth of 128GB/sec. Fig.1b outlines a naïve GpuRoofline model for NVIDIA GPU with peak performance of 1.03TFlopps/sec and peak bandwidth 144GB/sec. The graph is log-log scale and sets an upper bound on the performance of GPU kernels. The max attainable performance equals to min {peak performance, peak bandwidth * Compute Intensity}.

As shown in Fig.1, the vertical purple dashed line represents the Compute Intensity of hardware, calculated by peak performance dividing peak bandwidth. Two vertical red dashed lines represent two kernels with different Compute Intensity: the left one which Compute Intensity smaller than hardware Compute Intensity called memory-bound kernel; and the right one which Compute Intensity larger than hardware Compute Intensity called instruction-bound kernel. As will be explained later, the hardware Compute Intensity suggests the level of difficulty to achieve peak performance.



(a) AMD HD5850 GPU (b) NVIDIA C2050 GPU

Fig. 1. Naïve GPURoofline for GPUs

As we see, we must build a unique GPURoofline for each of the different GPU architecture. Fortunately, given a GPURoofline, we can use it repeatedly on different kernels.

3.2 Threshold Optimizations

We introduce Little's Law to guide our designs on communication. We also define the three components included in Little's Law: memory access latency, concurrency and the utilization of the peak bandwidth. The utilization of the peak bandwidth will drop if Little's is not satisfied.

Optimization Space. According to Little's Law, we defined optimization spaces as follows:

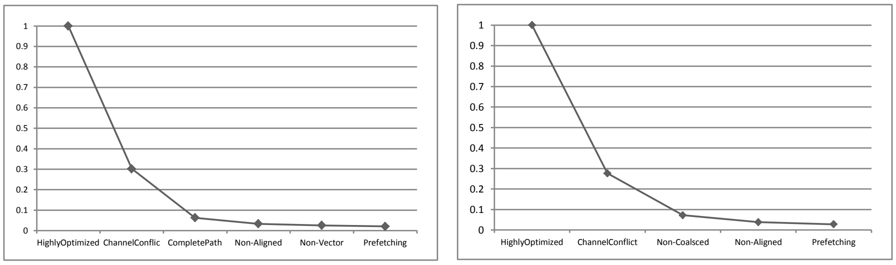
Eliminating Channel Conflict (ECC), just as local memory, global memory is divided into 8 partitions of 256-byte width on both AMD and NVIDIA GPU. Channel conflict occurs when concurrent global memory access requests queue up at some partitions while other partitions go unused. Rearrange data structure to ensure adjacent work-items access adjacent memory address is a common optimization technique.

Reducing Memory Transactions (RMT), coalescing global memory access requests into as few memory transactions as possible. Alignment, vector and coalesced access are the main methods to achieve this.

Using Software Prefetching (USP), the highest performance usually requires keeping many memory operations in flight, which is easier to do via prefetching than by waiting until the data is actually requested by the program.

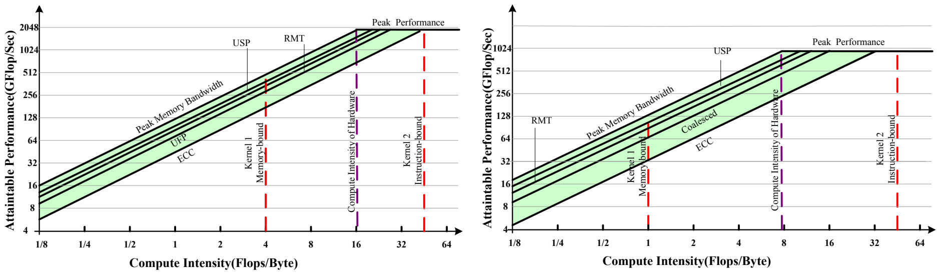
Using FastPath (UFP), this is for AMD GPU specially. Examine the code to ensure you are using FastPath not CompletePath, can improve performance significantly.

Threshold Carving. In this section, we will perform a sensitivity analysis to examine the impact of optimization methods on performance .We design a highly optimized implementation of copy micro-benchmark which the utilization



(a) AMD HD5850 GPU (b) NVIDIA C2050 GPU

Fig. 2. Performance changes along with the optimizations removed one by one



(a) AMD HD5850 GPU (b) NVIDIA C2050 GPU

Fig. 3. GPURoofline model with threshold carvings

of peak bandwidth can achieve 90% on both NVIDIA and AMD GPU. And then remove those optimization methods one by one in a particular order, Fig.2 shows performance changes.

From Fig. 2, we can see that for both NVIDIA and AMD GPU, the most important optimization method is eliminating channel conflict which was ignored in previous work. However, the second important optimization method is different: using FastPath for AMD GPU and coalesced access for NVIDIA GPU, respectively. Changing access patterns to allow data alignment is also important for both NVIDIA and AMD GPU. We add those optimization methods to our GPURoofline model:

As shown in Fig. 3, similar to performance changes, as we remove these optimization methods, new bandwidth curves will be formed below the peak bandwidth curve. We call these interior GPURoofline-like structures Memory Carvings. These Carvings not only provide some reasonable bounds on performance but also provide some suggestions for the optimizations. You cannot break through a ceiling without performing the associated optimization method first, so these memory carvings are called threshold carvings. We rank the Carvings from bottom to top as the order we remove the optimization methods.

3.3 Tradeoff Optimizations

We also introduce Little's Law to guide our designs on computation and define the three components included in Little's Law: concurrency, latency and throughput of effective instruction. Performance will drop if Little's is not satisfied.

Optimization Space. According to Little's Law, we defined optimization spaces as follows:

Reducing Dynamic Instructions (RDIS), increase the efficiency of instruction stream. There are four methods for this: minimizing divergent threads within a warp or a wavefront; eliminating common subexpression; loop-invariant code motion and loop unrolling. However, these optimizations must be balanced against the increased usage of hardware resources.

Instruction Selection Optimizations (INS), throughputs of GPU instructions are very different. Selecting instructions with lower latency as much as possible is a very desirable method for instruction-bound kernels.

Increasing Thread-level Parallelism (TLP), GPUs hide latency based on a large number of threads. Exploiting TLB, providing enough threads for each compute unit is a basic optimize method for GPUs.

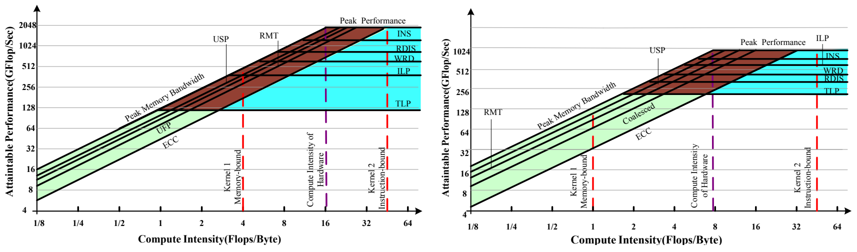
Increasing Instruction-level Parallelism (ILP), ensure the availability of independent instructions within a thread. This is usually achieved by loop unrolling, reordering the code and using vector instructions.

Work-redistribution (WRD), redistribute workloads across threads when there are workload imbalance. We can achieve it through four techniques: persistent thread, global queue, local queue and task stealing.

Tradeoff Carving. Using a similar analysis discussed in section3.2, we obtain some new GpuRoofline-like Carvings below GpuRoofline called Compute Carving. However, because of the discontinuous of optimization spaces, it is not clear that one should maximize or minimize an optimization method. So the Compute Carving is called tradeoff carving which just provides the insights into the performance improvement but not accurately, this is very different from the Memory Carving. The desired of accurate Compute Carving is the future work.

As shown in Fig. 4, when the Compute Intensity of a kernel greater than 0.81 for AMD GPU or 1.8 for NVIDIA GPU(calculated by hardware Compute Intensity divides process elements per stream core, then divides instruction cycles) we should consider the optimization of computation. We can also conclude from Fig 4 that, computation optimization for AMD GPU is more difficult than NVIDIA GPU, That is because AMD GPU is vector architecture and we cannot translate all the scalar instructions into vector instructions with appropriate length.

As Show in Fig.4, for both NVIDIA and AMD GPU, the most effective method is to exploiting TLP to hide latency. Exploiting ILP is the most obvious difference in the process of optimizing which is more effective for AMD GPU than NVIDIA GPU, because of AMD GPU's vector architecture. Additionally, RDIC is also an



(a) AMD HD5850 GPU (b) NVIDIA C2050 GPU

Fig. 4. GPURoofline model with tradeoff carvings

important optimization method. However, we must balance against the increased usage of hardware resources. Using Work-redistribution to enable load balance among threads can improve performance significantly on both NVIDA and AMD GPU for the irregular-parallel algorithm.

3.4 Data Locality

The main purpose of data locality is to increase kernel Compute Intensity. By increasing data reuse and decreasing the traffic of off-chip memory, this approach can improve performance significantly, especially for memory-bound kernel. Like memory access and computation constrained performance through performance carving, Compute Intensity also constrain performance like a wall, is called Compute Intensity Wall. We cannot achieve higher performance without improving kernel Compute Intensity especially for memory-bound kernels. So when you use GPURoofline model to guide your optimization and the performance is not achieve your expectation, the first optimization method you should think is increasing kernel Compute Intensity through data locality.

3.5 Interaction with Program Optimization

According to the GpuRoofline model, we can optimize kernels easily according to four rules:

Firstly, the Compute Intensity of a kernel determines the optimization region, and thus which optimization method to try. As shown in Fig.4, if the kernel dashed line falls into the green area, programmers should work only on the memory optimizations. If the dashed line falls into the blue area, programmers should work only on the computation optimizations. If the dashed line falls into the brown area, programmers should try both types of optimizations.

Secondly, optimization carvings suggest the corresponding methods that programmer should perform. And the gap between them represents the potential (Memory Carving) or relative potential (Computation Carving) benefits of related optimization method.

Thirdly, the order of the optimization carving suggests the optimization order.

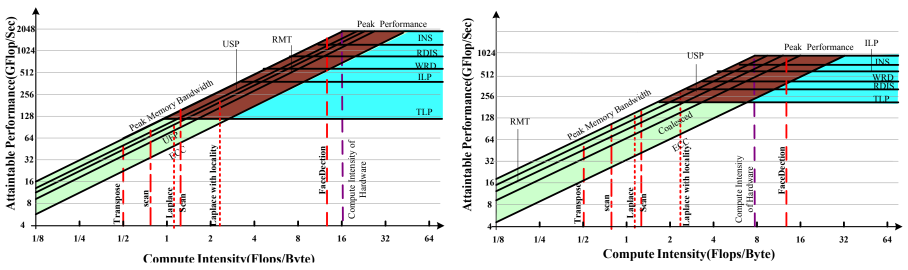
Finally, the ridge point marks the minimum Compute Intensity required to achieve peak performance.

4 Evaluation

In this section we demonstrate the usage of GPURoofline model through four kernels with different Compute Intensity. Table 1 shows the configuration of GPUs in our experiments in detail. Fig.5 shows optimization regions of these four kernels in GpuRoofline model. Note that, when calculating kernel Compute Intensity, we consider all the calculations, including address calculations.

Table 1. Configuration of the GPUs in our experiments

GPU	Clock Rate	PE	CU	Peak performance	Memory	Peak BW	Regisgers/CU	LDS/CU
AMD HD5850	0.725GHZ	288	18	2090GFlops	1.0GB	128GB/s	16K	32K
NVIDIA C2050	1.15GHZ	448	14	1030GFlops	3.0GB	144GB/s	16K	48K



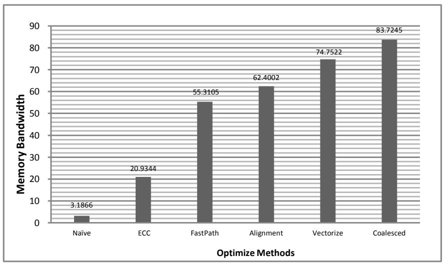
(a) AMD HD5850 GPU (b) NVIDIA C2050 GPU

Fig. 5. Optimization regions of these four kernels in GpuRoofline model

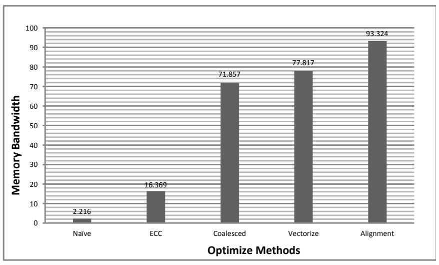
4.1 Matrix Transpose

In this section, we optimize matrix transpose under the guidance of GpuRoofline model. The transpose operation of each element performs two address calculations, and each address calculation performs 2 floating-point operations, so the compute intensity of matrix transpose is 2*2/8=0.5. According to optimization chain, our optimization work should only focus on the off-chip memory bandwidth optimizations.

As our GPURoofline model suggests, for both NVIDIA and AMD GPU, the first method to consider is eliminating channel conflict, and we achieve it by using a technique called Diagonal Block Reordering method. We also use vector memory access pattern to exploit ILP and data alignment to reduce memory transactions. In addition, we use local memory to make its global memory access pattern coalesced. Fig.6 shows the performance results when satisfies desired optimization methods one by one. The performance of this kernel is on a 2560 * 2560 matrix of float and uses memory bandwidth as the performance metric.



(a) AMD HD5850 GPU



(b) NVIDIA C2050 GPU

Fig. 6. Performance changes when satisfies optimization methods one by one

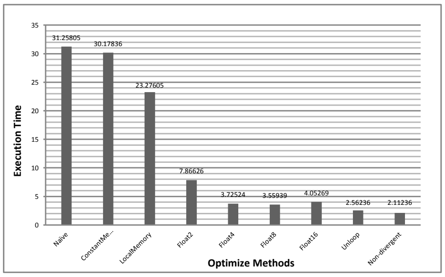
As shown in Fig. 6, eliminating channel conflict and using FastPath are the first two optimization methods for AMD GPU. However, for NVIDIA GPU, the first two optimize methods are eliminating channel conflict and coalesced memory access pattern. We can also see that, optimization on NVIDIA GPU is a little easier than AMD GPU. Using GPURoofline, the utilization of peak bandwidth achieves 65.4% and 64.8% on AMD GPU and NVIDIA GPU respectively.

4.2 Laplace Transform

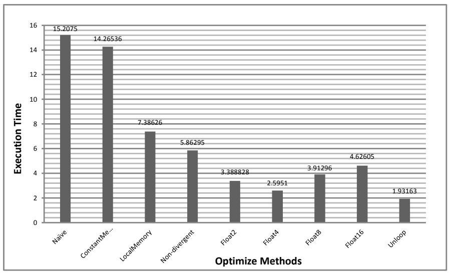
According to Laplace Transform algorithm, the transform of each element needs to perform 9 add and multiply operations. In addition, it requires 9 iterations and 10 address calculations. Each calculation contains two floating-point operations. So the Compute Intensity is 47/36=1.3. However, with this Compute Intensity, we can't obtain a satisfied performance. So we consider to use data locality. If the work-group size is 16*16, calculating these 256 elements need to transfer17*17 = 289 elements from off-chip memory to local memory. Furthermore, we put the Laplacian matrix into the constant memory, further reduces the dependence of the off-chip memory bandwidth. After data locality, the compute intensity of this kernel reaches to 3.2. Just as shown in Fig 5.

According to GPURoofline model, optimization works should focus on both memory and computation optimization. Fig.7 shows performance results when satisfies desired optimization methods one by one. The performance of this kernel is on a 1024 * 1024 matrix of float and uses execution time as the performance metric.

As shown in Fig. 7, data locality is a common optimization method for memory-bound kernels. Using vector instruction to exploit ILP can improve performance significantly for both NVIDIA and AMD GPU, however, when the vector length exceed a value, 8 for AMD GPU and 4 for NVIDIA GPU respectively, the performance decreases. This is because vector instructions need more register files, limits the number of threads that can be executed simultaneously. We can also see that, exploiting ILP is more efficient for AMD GPU than NVIDIA GPU. Reduce dynamic instructions through eliminating divergent and



(a) AMD HD5850 GPU



(b) NVIDIA C2050 GPU

Fig. 7. Performance changes when satisfies optimization methods one by one

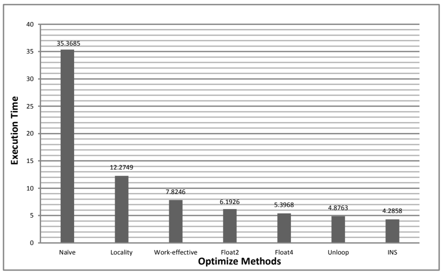
loop unrolling, are also efficient methods. Using GpuRoofline, the performance improved by 14.1 and 7.8 times on AMD GPU and NVIDIA GPU respectively.

4.3 Integral

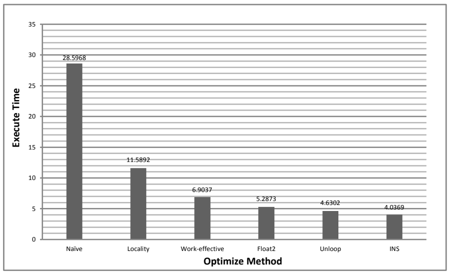
According to our implementation of integral algorithm, the Compute Intensity of this kernel is 4.2 after using locality as we have to execute so many memory address calculation and iterations.According to GPURoofline model, optimization works should focus on both memory and computation optimization.

In order to improve the efficiency of instructions, we use a more work-efficient parallel scan algorithm that performs $O(\mathrm{n})$ operations instead of a naïve version that performs $O(\mathrm{nlog2n})$ operations. We also optimize this kernel a step further under the guidance of GPURoofline. Fig.8 shows the performance results when satisfies desired optimization methods one by one. The performance of this kernel is on a 1024 * 1024 matrix of float and uses execution time as performance metric.

As shown in Fig.8, data locality optimization is the most important for memory-bound kernels. Work-effective scan algorithm can improve performance by improving the utilization of thread. As we discussed previously, because of AMD GPU's vector architecture, exploiting ILP can improve performance more



(a) AMD HD5850 GPU



(b) NVIDIA C2050 GPU

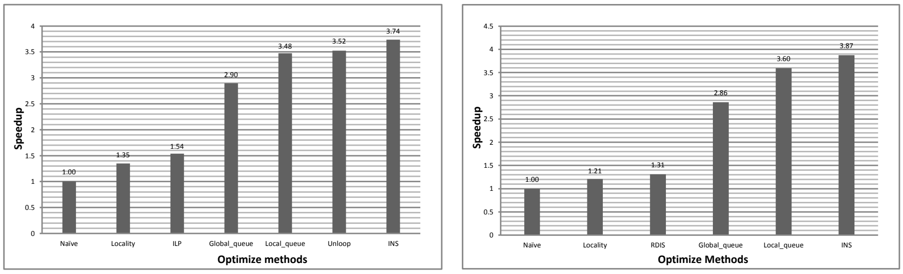
Fig. 8. Performance changes when satisfies optimization methods one by one

than NVIDIA GPU. Using GpuRoofline, the performance improved by 14.8 and 8.1 times on AMD GPU and NVIDIA GPU respectively.

4.4 FaceDetection

In this section, we optimize Viola-jones based face detection algorithm on GPUs according to GPURoofline. In this paper, our face detection kernel is the kernel that using cascade classifier to detect face. As shown in Fig.5, face detection kernel has high Compute Intensity, to 14.2 according to our implementation. According to GPURoofline, optimization work should focus on improving computation performance.

Different from algorithms discussed above, the face detection kernel is an irregular-parallel algorithm. There are serious load imbalance among threads. So we should use work-redistribution to address this problem. We use speedup to the naïve implementation as performance metric. Fig.9 shows the performance results when satisfies desired optimization methods one by one.



(a) AMD HD5850 GPU (b) NVIDIA C2050 GPU

Fig. 9. Performance changes when satisfies optimization methods one by one

As shown in Fig.9, Work-redistribution is the most effective optimize method for this kernel. In additional, using data locality to increase Compute Intensity and selecting instructions with higher throughput such as mad24 can also improve performance. Because face detection kernel is hard to vectorize, increasing ILP, mainly through reordering the code, there is no effect for NVIDIA GPU, although there is little effect for AMD GPU.Using GpuRoofline, the performance improved by 3.74 and 3.87 times on AMD GPU and NVIDIA GPU respectively.

5 Conclusion

We have presented GPURoofline, an empirical model for guiding performance optimizations on both NVIDIA and AMD GPU platforms. The goal is to help non-expert programmers with limited knowledge of GPU architectures implement high performance GPU kernels. Programmers can identify performance bottleneck and select appropriate optimization methods. Furthermore, we have observed that for best performance, optimization strategies are closely related to hardware architectures. Although the model is not designed to achieve perfect accuracy, it captures primary performance characteristics of GPUs.

We also demonstrated the usage of the model through four kernels with different compute densities. Experimental results show that under the guidance of the GPURoofline, performance of those kernels achieves 3.74∼14.8 times speedup compared to their naïve implementations on both NVIDIA and AMD GPU platforms.

Acknowledgements. We would like to thank reviewers for their helpful comments to our work. This work is supported by the National High-tech R&D Program of China (No. 2012AA 010902, No. 2012AA010903), the National Natural Science Foundation of China (No. 61133005, No.61100066) and ISCAS-AMD Fusion Software Center. Dr. Guoping Long is supported by National Natural Science Foundation of China (Grant No. 61100072).

References

1. Zhang, Y., Owens, J.D.: A quantitative performance analysis model for GPU architectures. In: High Performance Computer Architecture, pp. 382–393 (February 2011)
2. Baghsorkhi, S., Delahaye, M., Patel, S.J., Gropp, W.D., Hwu, W.-M.W.: An Adaptive Performance Modeling Tool for GPU Architectures. In: Principles and Practice of Parallel Programming, pp. 105–114 (January 2010)
3. Daga, M., Scogland, T.R.W., Feng, W-C.: Architecture-Aware Optimization on a 1600-core Graphics Processor. Technical Report TR-11-08, Computer Science, Virginia Tech.
4. Kothapalli, K., Mukherjee, R., Rehman, M.S., Patidar, S., Narayanan, P.J., Srinathan, K.: A performance prediction model for the CUDA GPGPU platform. In: International Conference on High Performance Computing, pp. 463–472 (2009)
5. Ryoo, S., Rodrigues, C.I., Stone, S.S., Baghsorkhi, S.S., Ueng, S., Stratton, J.A.: Program Optimization Space Pruning for a Multithreaded GPU. In: International Symposium on Code Generation and Optimization, pp. 195–204 (April 2008)
6. Hong, S., Kim, H.: An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In: International Conference on Computer Architecture, pp. 152–163 (2009)
7. Jang, B., Do, S., Pien, H.: Architecture-Aware Optimization Targeting Multithreaded Stream Computing. In: Second Workshop on General-Purpose on Graphics Processing Units (2009)
8. Meng, J., Morozov, V.A., Kumaran, K., Vishwanath, V., Uram, T.D.: GROPHECY: GPU Performance Projection from CPU Code Skeletons. In: Conference on High Performance Computing (2011)
9. Bauer, M., Cook, H., Khailany, B.: CudaDMA: optimizing GPU memory bandwidth via warp specialization. In: Conference on High Performance Computing(Supercomputing) (2011)
10. Govindaraju, N.K., Larsen, S., Gray, J., Manocha, D.: A Memory Model for Scientific Algorithms on Graphics Processors. In: ACM/IEEE Conference on Supercomputing (November 2006)

11. Williams, S., Waterman, A., Patterson, D.: Roofline: An Insightful Visual Performance Model for Multicore Architectures. Communications of the ACM, 65–76 (2009)
12. Lazowska, E.D., Zahorjan, J., Scott Graham, G., Sevcik, K.C.: Quantitative System Performance: Computer System Analysis using Queueing Network Models. Prentice-Hall. Inc., Upper Saddle River (1984)
13. Fatahalian, K., Sugerman, J., Hanrahan, P.: Understanding the Efficiency of GPU Algorithms for Matrix-matrix Multiplication. In: Conference on Graphics Hardware, pp. 133–137 (August 2004)
14. Taylor, R., Li, X.: A Micro-benchmark Suite for AMD GPUs. In: International Conference on Parallel Processing Workshops, pp. 387–396 (2010)
15. Liu, W., Muller-Wittig, W., Schmidt, B.: Performance Predictions for General-Purpose Computation on GPUs. In: International Conference on Parallel Processing, pp. 50–57 (September 2007)
16. Viola, P., Jones, M.: Robust Real-time object Detection. In: Second International Workshop on Statistical and Computation, pp (July 2011)