# Coordinate Strip-mining and Kernel Fusion to Lower Power Consumption on GPU

Guibin Wang
National Laboratory for Parallel and Distributed Processing
School of Computer, National University of Defense Technology
Changsha, Hunan, China
Email: wgbljl@gmail.com

***Abstract*—Although general purpose GPUs have relatively high computing capacity, they also introduce high power consumption compared with general purpose CPUs. Therefore low-power techniques targeted for GPUs will be one of the most hot topics in the future. On the other hand, in several application domains, users are unwilling to sacrifice performance to save power. In this paper, we propose an effective kernel fusion method to reduce the power consumption for GPUs without performance loss.**

**Different from executing multiple kernels serially, the proposed method fuses several kernels into one larger kernel. Owing to the fact that most consecutive kernels in an application have data dependency and could not be fused directly, we split large kernel into multiple slices with strip-mining method, then fuse independent sliced kernels into one kernel. Based on the CUDA programming model, we propose three different kernel fusion implementations, with each one targeting for a special case. Based on the different strip-ming methods, we also propose two fusion mechanisms, which are called invariant-slice fusion and variant-slice fusion. The latter one could be better adapted to the requirements of the kernels to be fused. The experimental results validate that the proposed kernel fusion method could effectively reduce the power consumption for GPU.**

***Keywords*-GPGPU, Kernel Fusion, Strip-mining, Power Efficiency**

## I. Introduction

Compared with general purpose CPUs, Graphics Processing Units (GPUs) integrate many simpler processing cores and provide extremely higher raw computing power. Therefore, modern general purpose processing systems tend to integrate GPU as the co-processor to accelerate typical computations. Although GPUs have relatively high power efficiency theoretically, there are still some key deficiencies limiting GPU's high capacity. In the CUDA programming model [1], a typical application is made up of a series of kernel calls. All the kernels have to be executed serially in the current implementation, which may results in low hardware utilization and results in imbalanced utilization along a series of kernel executions owing to the fact that different kernels may stress different hardware resources.

To address this problem, we propose an effective method of coordinating strip-mining [2] and kernel fusion to improve the power efficiency on GPUs. The kernel fusion method means merging two or more independent kernels into one larger kernel. Firstly, we introduce the potential benefit of kernel fusion on energy optimization. Generally, the relationship of dynamic power consumption $P$ and operating frequency $F$ could be described as $P = KF^{\gamma}$ (assuming $\gamma = 3$), where $K$ is an architecture-dependent constant. Observe that $P$ is a convex function of $F$. Therefore, with the same execution time, fusing two or more independent kernels could reduce the total energy consumption. The detailed verification could be found in Section III-B. Owing to the fact that most consecutive kernels in an application have data dependency and could not be fused directly, so firstly we apply strip-mining method to split a large kernel into multiple slices, then fuse independent sliced kernels into one kernel. The term slice means a segment of the whole stream referred in a kernel function.

The rest of the paper is arranged as follows. We introduce the implementation of kernel fusion in Section II. The algorithm of kernel fusion is discussed in Section III. The detailed experimental analysis is introduced in Section IV. Related work is given in Section V. We conclude our work in Section VI.

978-3-9810801-7-9/DATE11/@2011 EDAA

## II. Implementation of Kernel Fusion

Modern GPU architectures support thousands of threads executing in parallel. In the CUDA programming model threads are organized in two levels. At the higher level, all threads executing the same kernel function form a grid. Each grid consists of several thread blocks, with each one has unique coordinates. In turn, each thread block is a three-dimensional array of threads, which is explicitly defined by programmers.

Basing on the two-level thread hierarchy, we propose three different fusion implementations: inner-thread, inner-block and inter-block fusion, as illustrated in Fig.1. As seen from the pseudo-code in Fig.1.A, the inner-thread fusion method merges the computation of two kernels into a single thread. If the kernels to be fused have different thread or block spaces, a branch statement should be inserted to distinguish different working spaces, which may result in imbalanced workloads among threads. Similar to the inner-thread fusion, the inner-block merges the computation of two kernels into a single block, as shown in Fig.1.B. On the contrary, in the inter-block fusion, the computation of different kernels is distributed among different blocks, as shown in Fig.1.C.

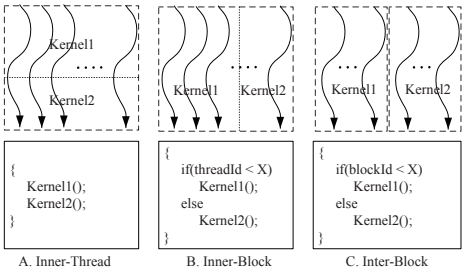


Fig. 1. Different fusion implementations (the dashed frame represent a thread block)

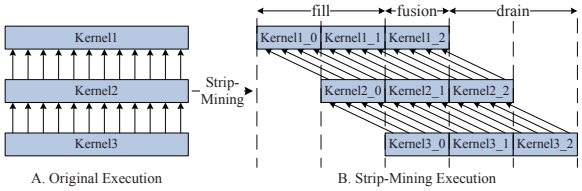


Fig. 2. Strip-mining based kernel fusion

## III. Coordinate Strip-mining and Kernel Fusion

Typically, the consecutive kernels in an application have data dependency and could not be fused directly. So, we first apply strip-mining method to split kernels into multiple slices, then fuse independent sliced kernels into a new kernel. Fig.2 illustrates the process of applying strip-mining method on a series of kernels with data dependence. In the original execution shown in Fig.2.A, the kernel2 need to access the output of kenrel1, and similar relationship exists between kernel3 and kernel2. So the kernel1 and kernel2 or kernel2 and kernel3 could not be fused directly. However, most GPU kernels have the relatively good space locality in the memory access, that is the computation only needs several nearest neighbor points, such as stencil operation. Benefitting from this feature, we could apply strip-mining method to split the whole kernel into multiple slices, as shown in Fig.2.B. Then skew the execution of the sliced kernels to get independent sliced kernels which could be used as the candidates for kernel fusion. As shown in Fig.2.B, the sliced kernel kernel1_2, kernel2_1 and kernel3_0 could be fused without violating data dependency.

In the following paragraphs, we will focus on determining the total number of slices and the size of each slice for the largest energy reduction.

### A. Determining the Maximum #Slices

In the current runtime implementation for GPU, there is an unnegligible startup overhead in launching a kernel program [3], therefore the strip-mining process increases the total kernel launch overhead. Considering kernel startup overhead, the execution time of a kernel program could be described as

$$T = T_{startup} + T_{comp}(l) = \alpha + \beta l$$

,where $\alpha$ represents the startup overhead, $l$ denotes the length of slice and $\beta$ is a kernel- and hardware-specific constant, which represents the average processing time for a single element.

Let $n$ and $m$ denote the number of slices and the number of kernels to be fused. In the serial execution, the total execution time $T_{serial} = m\alpha + \sum_{j=1}^{m} \beta_j L$, where $L$ represents the size of the whole output data stream. Assuming the fused kernel have the same computation time with serial execution. So, in the strip-mining execution the total execution time $T_{fuse} = (n - (m-1) + m(m-1))\alpha + \sum_{j=1}^{m} \beta_j L$. Assuming a performance relax factor $\delta$, which could be defined as $\delta = \frac{T_{fuse} - T_{serial}}{T_{serial}}$. Under a user-determined factor $\delta$, the maximum number of slices should be

$$n_{max} = \lfloor \frac{\delta T_{serial}}{\alpha} - (m-1)^2 + m \rfloor$$

### B. Determining the Size of Slices

The key problem in strip-mining is to decide the size of each slice. A straightforward method is to make equal-size slices, which is called *invariant-slice* fusion in this paper. In the invariant-slice fusion, the workload in each the fusion stage does not change, which results in that all the kernels must have the same slice sizes. On the contrary, a more effective method should consider the computing/memory ratios of the kernels to be fused and determine the slice sizes of each kernels to maximize the power efficiency in the fused kernel. While this fusion method results in variant workload in each fusion stage, which is called *variant-slice* fusion. In the following paragraphs, we will discuss both methods in detail.

To facilitate the explanation, we give the related symbols first. Let $c_i$ and $m_i$ denote the computation cycles and memory cycles in producing single element for the kernel $i$ ($i = 1, 2$); $F_c$ and $F_m$ are the maximum operating frequencies of shader core and memory units respectively and the corresponding constant coefficients in the power equation are $K_c$ and $K_m$ respectively. Note that in this paragraph, we do not consider the startup overhead, which is covered in the above section and only focus on the computing time.

**Invariant-Slice Fusion**

Fig.3 illustrates an invariant-slice fusion. As shown in the figure, all the fused kernels have the same slice size. Without kernel fusion, the execution time of running one slice of kernel $i$ ($i = 1, 2$) is $T_{slice,i} = \frac{max\{c_i, m_i\}}{F_c} l$ and the total energy consumption with DVFS is

$$\begin{aligned} E_{slice,i} =& T_{slice,i}(P_{core} + P_{mem}) \\ =& T_{slice,i}(K_c(\frac{c_i}{F_c T_{slice,i}} F_c)^3 + K_m(\frac{m_i}{F_c T_{slice,i}} F_m)^3) \\ =& \frac{K_c c_i^3 + K_m \frac{F_m^3}{F_c^3} m_i^3}{T_{slice,i}^2} l \end{aligned}$$

It is not difficult to find that $max\{c_1 + c_2, m1 + m2\} \leq max\{c1, m1\} + max\{c2, m2\}$, so the execution time for the fused kernel is no more than that when executing two kernels serially. Under the same execution time with serial execution, the total energy consumption for the fused kernel becomes

$$E_{slice,1+2} = \frac{K_c(c_1 + c_2)^3 + K_m \frac{F_m^3}{F_c^3}(m_1 + m_2)^3}{(T_{slice,1} + T_{slice,2})^2} l$$

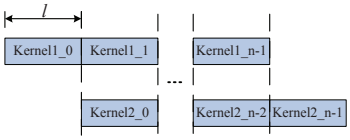


Fig. 3. Invariant-slice Fusion

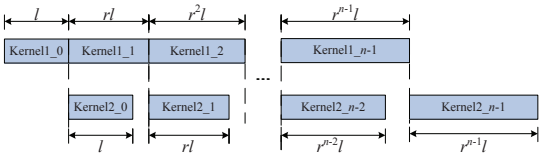


Fig. 4. Variant-slice Fusion

According to the theorem in [4], the following inequality holds,

$$\frac{(c_1+c_2)^3}{(T_{slice,1}+T_{slice,2})^2} \leq \frac{c_1^3}{T_{slice,1}^2} + \frac{c_2^3}{T_{slice,2}^2}$$

, so as the second term in $E_{slice,i}$ and $E_{slice,1+2}$. Then we could get

$$E_{slice,1+2} \leq E_{slice,1} + E_{slice,2}.$$

Therefore, under the same execution time, the fused kernel consumes no more energy than the one with serial execution.

Corresponding to the Fig.3, the total energy consumption becomes

$$E_{fuse} = E_{slice,1} + E_{slice,2} + (n-1)E_{slice,1+2}$$

Considering $l = \frac{L}{n}$, we can see that $E_{fuse}$ decreases with $l$. Therefore, for the optimal energy consumption, we should choose the minimum slice size. From the discussion in III-A, under a given performance relax factor, the slice size $l = \frac{L}{n_{max}}$ for all the fusion stages.

**Variant-Slice Fusion**

Different from invariant-slice fusion, variant-slice fusion has variational slices during different fusion stages. As shown in Fig.4, the ratio of slice sizes between consecutive fused kernels is kept $r$. For the $j$th slice, the original execution time is

$$\begin{aligned} T'_{j,1+2} &= \frac{max\{r^j c_1 l, r^j m_1 l\} + max\{r^{j-1} c_2 l, r^{j-1} m_2 l\}}{f_c} \\ &= (rT'_{slice,1} + T'_{slice,2}) r^{j-1} \end{aligned}$$

,where $T'_{slice,i}$ denotes the execution time of the first slice for kernel $i$.

Under the same execution time, the energy consumption of $j$th fused kernel is

$$E'_{j,1+2} = \frac{K_c(rc_1+c_2)^3 + K_m \frac{F_m^3}{F_c^3}(rm_1+m_2)^3}{(rT'_{slice,1} + T'_{slice,2})^2} r^{j-1} l$$

And the total energy consumption via kernel fusion becomes

$$\begin{aligned} E'_{fuse} =& E'_{slice,1} + r^{n-1} E'_{slice,2} + (E'_{1,1+2} + \cdots + E'_{n-1,1+2}) \\ =& E'_{slice,1} + r^{n-1} E'_{slice,2} + \frac{1-r^{n-1}}{1-r} E'_{1,1+2} \end{aligned}$$

,where $E'_{slice,i}$ denotes the energy consumption of the first slice for kernel $i$. With certain $n_{max}$ determined in Section III-A, the energy consumption $E'_{fuse}$ could be viewed as a function of factor $r$. It is clear that $E'_{fuse}$ is minimized when $\frac{\partial E'_{fuse}}{\partial r} = 0$ ,which could be simply solved using existing mathematic tool. Let $r_{min}$ denote the variant slice factor for minimum $E'_{fuse}$, therefore, the size of the first slice $l$ is $\frac{1-r_{min}}{1-(r_{min})^{n_{max}}} L$.

## IV. Experiments

### A. TestBed and Benchmarks

We use an in-house GPU simulator in the experiments. The hardware configuration is similar to NVIDIA's QuadroFX5800 GPU, which has 30 shader cores and 102GB/s peak off-chip bandwidth. The operating frequency of shader core varies from 500MHz to 1.3GHz with the step of 100MHz.

We primarily evaluate the proposed method using a microbenchmark, which consists of two kernels: one is an extremely compute-intensive kernel, termed Compute kernel, and the other one is an extremely memory-intensive kernel, termed Memory kernel. Besides the microbenchmark, we also choose two real applications Swim [5] and Srad [6] in the experiments.

### B. Evaluation

Fig.5 gives the execution time and energy consumption for the microbenchmark with #slice being 10. With DVFS on single kernel respectively, the execution time increases by 1.8% and the energy consumption reduces by 15.2%. However, through kernel fusion the total execution time reduces 11.2% of the one with serial execution and the energy consumption reduces to 97.6% even without DVFS. After relaxing execution time to the serial one, the energy consumption of the fused kernel reduces by 34.9%. Fig.6 gives the relative energy consumption with varied #slice. All the results are normalized to the serial one. The result shows that the total energy consumption reduces as the #slice increases, however the percentage of energy reduction does not improve significantly after the #slice being larger than 10, which demonstrates that commonly a moderate #slice is enough for energy optimization. Fig.7 illustrates the comparison in energy consumption between invariant- and variant-slice fusion. The result shows that variant-slice fusion always performs better than invariant-slice one. However, the difference is smaller than the theoretical result. One reason is the core voltage follows sub-linear relationship with the operating frequency in the simulation configuration, implicitly reducing the benefit from DVFS.

Fig.8 illustrates the core frequency for the original and fused kernels in Swim with DVFS-enabled. The result shows that the frequency changes among different kernel executions. Through kernel fusion, the operating frequency could be smoothed. Although fusing two memory-intensive kernels could not improve performance, it does reduce the energy consumption. Fig.9 gives the energy consumption for the fused kernel relative to the serial one with DVFS-enabled. The term

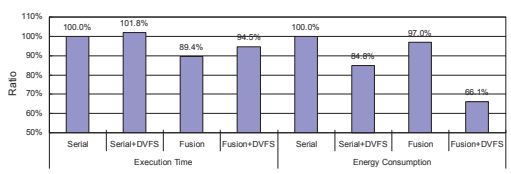


Fig. 5. Comparison of execution time and energy consumption for the microbenchmark (#slice=10)

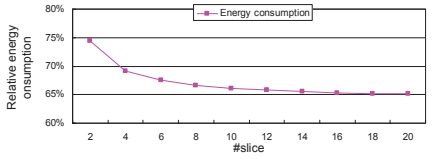


Fig. 6. Energy consumption with varied #slices for the microbenchmark

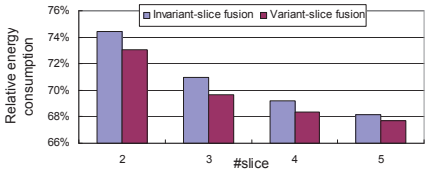


Fig. 7. Comparison of invariant- and variant-slice fusion

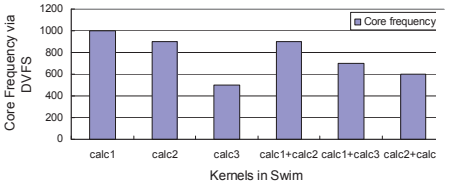


Fig. 8. The core frequency in Swim

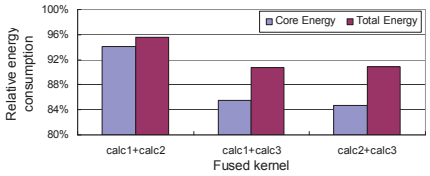


Fig. 9. Energy consumption of the fused kernels in Swim

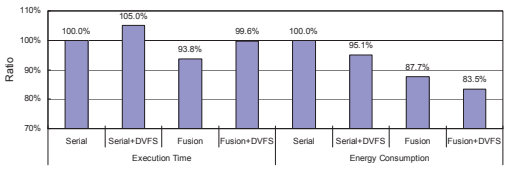


Fig. 10. Execution time and energy consumption of Srad

"core energy" represents the energy consumption of processing core, while the term "total energy" additionally includes the energy consumption in memory system. Relatively speaking, calc1 has the highest computation-to-memory ratio and calc3 has the lowest one among the three kernels. Therefore, fusing calc1 or calc2 with calc3 achieve relatively higher energy reduction.

Fig.10 gives the execution time and energy consumption for Srad. It is interesting that even without DVFS, the energy consumption of fused kernel is smaller than that of the serial execution with DVFS-enabled. After examining the simulation profile, we find that the number of instructions in fused execution is smaller than that of serial execution. One reason for this is that the computation about array index and the access to the thread or block ID register is reduced by half in the fused kernel. This is a potential benefit from kernel fusion. Under the same execution time, the energy consumption of the fused kernel reduces by 16.5% compared with the serial one.

## V. Related Work

As GPU comes into general purpose computing field and high performance computing field. Low-power optimization technique are concerned by the manufacturers and researchers. Huang [7] compared performance and energy consumption of CPU and GPU with an biological code and pointed out that GPU's high power efficiency have much relationship with the optimization strategy and application features. Collange analyzed the impact of multiprocessor and memory hierarchy on power consumption [8]. The most recent study by Hong [9] established an integrated performance and power model which could be used to predict the optimal number of active cores reaching the highest power efficiency.

Marisabel [10] proposed kernel-level parallelism to concurrently execute multiple kernels which underutilize GPU processing resources. Different that, we mainly focus on energy consumption via kernel fusion and validate that the proposed method could also be applied even for the kernels which could fully make use of all the processing cores.

## VI. Conclusion

This paper proposes the kernel fusion method to improve the power efficiency without sacrificing performance. Through detailed experimental evaluation, the proposed kernel fusion method could effectively reduce power consumption and improve power efficiency for GPU architecture. As the GPUs support DVFS at runtime, we will evaluate the kernel fusion method on real system and implement a automatic tool to perform the necessary code transformation.

## Acknowledgment

This work is supported by the NSF of China under Grant No. 60921062, No. 60903059 and No. 60903044.

## References

[1] NVIDIA, "Compute unified device architecture programming guide v2.1beta," 2009.

[2] A. Das, W. J. Dally, and P. Mattson, "Compiling for stream processing," in *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM Press, 2006, pp. 33–42.

[3] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *SC*. IEEE/ACM, 2008, p. 31.

[4] T. Li and C. Ding, "Instruction balance and its relation to program energy consumption," in *LCPC'01: Proceedings of the 14th international conference on Languages and compilers for parallel computing*. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 71–85.

[5] G. Wang, T. Tang, X. Fang, and X. Ren, "Program optimization of array-intensive spec2k benchmarks on multithreaded gpu using cuda and brook+," *International Conference on Parallel and Distributed Systems*, vol. 0, pp. 292–299, 2009.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. ha Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009*. IEEE, 2009, pp. 44–54.

[7] S. Huang, S. Xiao, and W. Feng, "On the energy efficiency of graphics processing units for scientific computing," in *Fifth Workshop on High-Performance, Power-Aware Computing (HPPAC'09)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–8.

[8] S. Collange, D. Defour, and A. Tisserand, "Power consumption of gpus from a software perspective," in *ICCS '09: Proceedings of the 9th International Conference on Computational Science*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 914–923.

[9] S. Hong and H. Kim, "An integrated gpu power and performance model," in *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2010, pp. 280–289.

[10] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron., "Enabling task parallelism in the cuda scheduler," in *Proceedings of the Workshop on Programming Models for Emerging Architectures (PMEA)*, Raleigh, NC, USA, 2009, pp. 69–76.