Data Transformations Enabling Loop Vectorization on Multithreaded Data Parallel Architectures

Byunghyun Jang, Perhaad Mistry, Dana Schaa, Rodrigo Dominguez, and David Kaeli
Department of ECE, Northeastern University, Boston, MA 02115 USA
{bjang,pmistry,dschaa,rdomingu,kaeli}@ece.neu.edu

Abstract

Loop vectorization, a key feature exploited to obtain high performance on Single Instruction Multiple Data (SIMD) vector architectures, is significantly hindered by irregular memory access patterns in the data stream. This paper describes data transformations that allow us to vectorize loops targeting massively multithreaded data parallel architectures. We present a mathematical model that captures loop-based memory access patterns and computes the most appropriate data transformations in order to enable vectorization. Our experimental results show that the proposed data transformations can significantly increase the number of loops that can be vectorized and enhance the data-level parallelism of applications. Our results also show that the overhead associated with our data transformations can be easily amortized as the size of the input data set increases. For the set of high performance benchmark kernels studied, we achieve consistent and significant performance improvements (up to 11.4X) by applying vectorization using our data transformation approach.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors-Code generation, Compilers, Optimization

General Terms Algorithms, Performance, Experimentation

Keywords Loop Vectorization, Data Transformation, GPGPU

1. Introduction and Memory Access Pattern Modeling

Data parallel architectures (e.g., Graphics Processing Units (GPUs)) achieve high performance by executing thousands of kernel instances on different data points using many cores in a SIMD fashion and by hiding memory latency using zero-overhead hardware thread switching. However, due to the underlying GPU memory architecture, applications can experience long stalls if they possess irregular memory access patterns. If a number of active threads try to access memory simultaneously, they may incur thread stalls due to serialized off-chip memory access [1–3]. Loop vectorization is a commonly used technique on vector machines to provide parallel memory access and avoid these stalls [2, 4].

Memory Access Pattern Modeling: We develop a mathematical model, based on a previously proposed model targeting locality on a single core [5], that captures array-based memory access patterns present in a loop nest. Consider a loop nest of depth D that accesses an M dimensional array, where D and M do not need to match (e.g., 2D matrix multiplication requires a three-level loop nest). The memory access pattern of the array in the loop is represented as a *memory access vector*, $\vec{m}$, which is a column vector of size M starting from the index of the first dimension. The memory access vector is then decomposed as an affine form:

$$\vec{m} = \mathbf{M}\vec{i} + \vec{o}$$

where $\mathbf{M}$ is a memory access matrix whose size is $M \times D$, $\vec{i}$ is an iteration vector of size D traversing from the outermost to the innermost loop, and $\vec{o}$ is an offset vector that is a column vector of size M and determines the starting access point in an array. Note that we consider only loops with accesses to arrays that are affine functions of the loop indices and symbolic variables. We have found that this restriction does not limit us since most scientific applications involve loops possessing affine access patterns [6].

Each column in the memory access matrix, $\mathbf{M}$, represents the memory access pattern of the corresponding loop level, and the number of columns dictates the depth of the loop nest. Each row in $\mathbf{M}$ and $\vec{o}$ represents the memory access pattern of each dimension of the array, and the number of rows dictate the array dimension.

2. Vectorization and Data Transformation Rule

The memory access matrix $\mathbf{M}$ of an array captures all memory access patterns in terms of each dimension of the array and each level of the loop nest. However, since vectorization requires the data being accessed to be located contiguously in memory, our interest is in the memory access pattern of the last dimension, that is, the last row in the memory access matrix. In order to vectorize multiple arrays together, each array needs to have the same memory access pattern with respect to its last dimension. Therefore, we define multiple arrays as *vectorizable* together if they all have same last row in their memory access matrices and offset vectors.

As compared to conventional vectorization techniques, our framework requires a new metric for quantifying vectorization since our vectorization methods are different. We define loops as *intrinsically vectorizable* if all arrays can be directly mapped to vector types without any changes to the loop body. We also define *fractionally vectorizable* if some, but not all, arrays can be mapped to vector types. In the fractionally vectorizable case, we use the equation below to further quantify the degree of vectorization (Note that Q_V of an intrinsically vectorizable loop is 100%):

$$Q_V(\%) = \frac{\#\ of\ Vectorizable\ Array\ Instances}{Total\ \#\ of\ Array\ Instances} \times 100$$

Each instance of an array in a loop nest is classified by its data access pattern; the data transformation rules required for vectorization are generated based on the above classification. The data

PPoPP'10, January 9–14, 2010, Bangalore, India.
ACM 978-1-60558-708-0/10/01.

Table 1. Summary of experimental results for vectorization on an AMD platform. $*$Vectorization speedup is relative to a scalar GPU baseline, with data transformation time included in the vectorized execution time - input vector size is 3072 elements for each dimension.

Benchmark	Memory Access Patterns	Data Transformations	Q_V	Speedup*	Overhead*
Livermore 19	Linear, Shifting, Reverse Linear	Duplicate, Shifting, Reversing Column	100%	11.4×	3.39%
NAS Cholesky	Linear, Non-unit Stride	Dimension Switch (3D)	100%	1.39×	1.36 %
NAS Vpenta	Linear, Non-unit Stride	Dimension Switch (2D)	100%	2.32×	23.16 %

transformation rule generated for each array is composed of a transformation matrix $\mathbf{T_m}$ of size $D \times D$ and a shift vector $\vec{t_v}$ of size D, denoted as the pair $(\mathbf{T_m}, \vec{t_v})$.

The application of $(\mathbf{T_m}, \vec{t_v})$ entails two changes. First, the actual data (i.e., the array layout) is transformed on the CPU using our data transformation routine, which takes as input the original data and transformation rules, and produces restructured data. Second, a new memory access vector, $\vec{m}'$ is generated as follows.

$$\vec{m}' = \mathbf{M}'\vec{i} + \vec{o}' = \mathbf{T_m}\mathbf{M}\vec{i} + (\mathbf{T_m}\vec{o} + \vec{t_v})$$

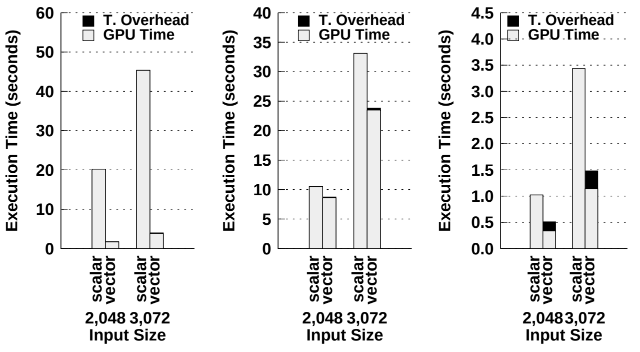
Access Pattern Classification and Transformation Rules: *Linear* and *reverse linear* memory access patterns refer to accesses in which a dimension of an array is contiguously accessed with respect to the iteration space. The data transformation required for these patterns is to reverse the layout of the data. A *shifted* pattern refers to an access in which a dimension of an array is contiguously accessed with respect to the iteration space (e.g., a linear pattern), but this contiguous access is shifted by some constant. The data transformation required for this pattern is to shift all array elements by a constant amount. An *overlapping* memory access pattern refers to an access in which a dimension of the array is accessed by more than one iteration space and the required transformation for this pattern is the same as in a shifted pattern, though with a dynamic shift amount. A *non-unit stride* memory access pattern refers to a pattern in which a dimension of an array is accessed with a non-unit stride. The data transformation needed for linearizing a non-unit stride pattern is to gather elements being accessed into a large chunk of linear addressable memory. A *random* memory access pattern includes instances where a dimension of an array is accessed in a random fashion. Memory accesses that are indexed by another array usually fall into this category. Linearization of a random memory access pattern remains an open problem for loop vectorization and we do not attempt to transform loops possessing random patterns in this work.

The actual access pattern representations, resulting algorithms for the vectorization test and data transformation rules are available in a technical report [7].

3. Experimental Results and Conclusion

We have evaluated the effectiveness of our proposed methodology using AMD's Stream Computing SDK 1.4 and Catalyst 9.2 graphic driver (which incorporates graphic compiler) on an AMD Radeon HD3870 (R670 family) GPU. The host system is an Intel Core 2 Duo 2.66 GHz CPU running 64-bit Linux with 2GB main memory. For each benchmark, two GPU versions (scalar and vector) are implemented using the Brook+ stream programming language [4]. We selected 3 loops from two high performance computing benchmark suites: Livermore Loops and NAS kernels. Table 1 and Figure 1 summarize our results. For the benchmark kernels studied, we can see that vectorization can significantly benefit performance (up to 11.4× speedup over the scalar version) and that the overhead associated with our data transformations is negligible.

We believe that our framework can be used to implement a number of other potential optimizations targeting massively multithreaded data parallel architectures such as memory coalescing and memory space selection (e.g., on NVIDIA GPUs).



(a) Livermore 19 (b) NAS Cholesky (c) NAS Vpenta

Figure 1. Performance comparison between scalar and vector implementations on an AMD platform.

Acknowledgments

We would like to thank the shader compiler group at AMD for their support and generous hardware donation for this work. This work was supported by the NSF ERC program, award EEC-9986821.

References

[1] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 1–10.

[2] B. Jang, S. Do, H. Pien, and D. Kaeli, "Architecture-aware optimization targeting multithreaded stream computing," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: ACM, 2009, pp. 62–70.

[3] B. Jang, D. Kaeli, S. Do, and H. Pien, "Multi GPU Implementation of Iterative Tomographic Reconstruction Algorithms," in *Biomedical Imaging: From Nano to Macro, 2009. ISBI 2009. 6th IEEE International Symposium on*, Jun 2009.

[4] AMD, "Brook+," May 2006, http://ati.amd.com/technology/streamcomputing.

[5] S. T. Leung and J. Zahorjan, "Optimizing data locality by array restructuring," University of Washington, Tech. Rep. TR 95-09-01, 1995.

[6] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: an analytical representation of cache misses," in *ICS '97: Proceedings of the 11th international conference on Supercomputing*. New York, NY, USA: ACM, 1997, pp. 317–324.

[7] B. Jang, P. Mistry, D. Schaa, R. Dominguez, and D. Kaeli, "Data Transformations Enabling Loop Vectorization, NUCAR Technical Report," Nov 2009, http://www.ece.neu.edu/groups/nucar/publications.html.