Exploiting Hyper-Loop Parallelism in Vectorization to Improve Memory Performance on CUDA GPGPU

Shixiong Xu, David Gregg

1 Software Tools Group, Department of Computer Science, Trinity College, The University of Dublin, Ireland
2 Lero, The Irish Software Engineering Research Centre
Email: \{xush, dgregg\}@scss.tcd.ie

Abstract—Memory performance is of great importance to achieve high performance on the Nvidia CUDA GPU. Previous work has proposed specific optimizations such as thread coarsening, caching data in shared memory, and global data layout transformation. We argue that vectorization based on hyper loop parallelism can be used as a unified technique to optimize the memory performance. In this paper, we put forward a compiler framework based on the Cetus source-to-source compiler to improve the memory performance on the CUDA GPU by efficiently exploiting hyper loop parallelism in vectorization. We introduce abstractions of SIMD vectors and SIMD operations that match the execution model and memory model of the CUDA GPU, along with three different execution mapping strategies for efficiently offloading vectorized code to CUDA GPUs. In addition, as we employ the vectorization in C-to-CUDA with automatic parallelization, our technique further refines the mapping granularity between coarse-grain loop parallelism and GPU threads. We evaluated our proposed technique on two platforms, an embedded GPU system — Jetson TK1 — and a desktop GPU — GeForce GTX 645. The experimental results demonstrate that our vectorization technique based on hyper loop parallelism can yield performance speedups up to 2.5× compared to the direct coarse-grain loop parallelism mapping.

Keywords—vectorization; hyper loop parallelism; thread coarsening; memory performance; CUDA GPU

I. INTRODUCTION

General Purpose Graphics Processing Units (GPGPUs), in particular, the Nvidia CUDA GPU, are widely used in a variety of machines. The deep hierarchy of both execution model and memory organization makes manually writing high performance code for the CUDA GPU error-prone and tedious. A long standing research goal has been to automatically generate GPGPU code from either auto-parallelization or compiler directive based languages (e.g. OpenMP [1], OpenACC [2]).

For parallel loops with data in an array of structures (AoS), directly mapping each loop iteration to a GPU thread may expose non-unit stride data access. When the data access pattern has unit stride, global memory accesses can be easily coalesced. However, non-unit stride accesses have a great impact on the effective memory bandwidth [3]. Therefore, optimizing non-unit stride memory access is of great importance to the performance of CUDA programs.

Non-unit stride data access is also an obstacle for efficient vectorization on CPUs. Apart from optimization with data reorganization instructions, super-word level parallelism (SLP) vectorization [4] [5] is deemed as an effective way to vectorize loops with data in AoS. However, to the best of our knowledge, dealing with non-unit stride memory access in C-to-CUDA with SLP vectorization has not been explored either in the research literature or in existing compilers. In this paper, we propose an improved SLP vectorization technique based on hyper loop parallelism to deal with the non-unit stride data access in C-to-CUDA. Hyper loop parallelism (HLP) vectorization is used to deal with semi-isomorphic sub-graphs of data flow graphs of a vectorizable loop and has proven effective on the CPU [6]. In this paper, we argue that efficient exploitation of hyper loop parallelism in vectorization can significantly improve the memory performance on the CUDA GPU. Moreover, the abstraction of hyper loops gives a way to further refine the mapping granularity between loop iterations and GPU threads.

In this paper, we put forward a compiler framework to efficiently exploit hyper loop parallelism in vectorization to improve the memory performance on the Nvidia CUDA GPU. Our approach consists of identification of hyper loop parallelism and efficient mapping to the GPU. We present a scheme to map conventional SIMD operations to the GPU. Based on this mapping scheme, we introduce a code generation technique to generate efficient CUDA code. Moreover, we examined three mapping strategies on how to efficiently offload vectorized code to the GPU. We implemented our technique on top of the Cetus source-to-source compiler [7] [8], which already contains a basic code generation framework from C to CUDA. The experimental results demonstrate that our HLP vectorization can yield performance speedups up to 2.5× compared to the direct
hyper loop

Hyper loop parallelism (HLP) vectorization consists of two phases, vectorization analysis and transformation.

1) Vectorization Analysis: Before HLP vectorization analysis, data dependence analysis is first applied to analyze whether a given loop is vectorizable or not. Data-flow analysis is also used to collect the downwards exposed definitions, which are classified into ordinary definitions and reduction definitions. The HLP vectorization analysis contains two steps, collect slices and group slices. The results of these two steps are shown in Fig. 2 and Fig. 3, respectively.

Slice grouping works similar to the super-word level parallelism (SLP) vectorization that tries to pack isomorphic instructions into groups for vectorization [4]. In contrast to the SLP vectorization, the grouping of slices starts from contiguous memory stores, and packs isomorphic operations from different slices. Either fully grouping or partially grouping is applied according to whether the computation structures of all the slices are the same or not. For the motivating example, because the computations at the node #0 and node #1 in Fig. 2 are not isomorphic, only partially grouping is employed. As illustrated in Fig. 3, two kinds of actions extract and merge are annotated on some edges to depict how the data flows between the connected nodes.

2) Vectorization Transformation: HLP vectorization transformation includes two phases, expand grouped slices and global SIMD lane optimization.

Expand Grouped Slices Each grouped DAG is transformed into a vectorized DAG with vector operations on virtual vectors with respect to a loop unrolling factor. The width of the virtual vector of each node is decided by multi-
plying the loop unrolling factor and the size of the node. For
the CUDA GPU, we choose the warp size as the width of
the physical vector. In the expansion, **SIMD lane descriptors**
are annotated to describe the patterns of SIMD lanes for
each node. **SIMD lane descriptors** are in the format of
id[start_position: size: stride], where id is the name of
an array, a pointer or a virtual vector, size is the
number of lanes, stride is the lane pattern. For the
grouped DAG in Fig. 3, the vectorized DAG after expansion
is shown in Fig. 4.

### Figure 4. Illustration of vectorization expansion with a loop unrolling factor 32. The CUDA warp size is 32.

**A. SIMD Vectors**

**Global SIMD Lane Optimization** The global SIMD lane-wise optimization tries to optimize the allocation of
SIMD lanes according to the changes of SIMD lanes be-
tween nodes by inserting new nodes corresponding to any
of the four SIMD lanes operations - pack, unpack, merge
and permute. Pack and unpack deal with the changes
of the vector size, merge performs blending of two vectors
with the given SIMD lane information, permute handles
the changes of ordering of SIMD lanes between two vectors
in the same size. The operations SwapEvenOddLanes and
MergeEvenOddLanes in Fig. 5 are concrete instances of
the permute and merge, respectively.

The global SIMD lane optimization consists of two
passes, a top-down pass and a bottom-up pass on the
expanded DAGs. The top-down pass tries to adjust the
widths of virtual vectors and SIMD lane patterns according
to the memory loads in the leaf nodes. For example, the node
#1 in Fig. 4 has a destination vector vtmp3 with the SIMD
lane pattern of [0:32:1]. The top-down pass changes
the SIMD lane pattern into [0:64:2] according to the
operand vtmp2 [0:64:2] because both operands have strided SIMD lane patterns. On the other hand, the bottom-
up pass propagates the SIMD lane information of the root
nodes to the leaf nodes and inserts the four SIMD lane
operations accordingly. The bottom-up pass, in particular,
takes care of the join nodes represented by Merge.

### III. Hyper Loop Parallelism on the CUDA GPU

**B. SIMD Operations**

**Virtual SIMD vectors via thread local variables** It is
intuitive that a thread local variable in a group can virtually
present a SIMD vector. According to the CUDA execution
model, all GPU threads are divided into blocks, and the
threads in a block are executed in batch, that is, a warp with
16 or 32 GPU threads. We choose the size of a warp as
the vector size for two reasons: 1) threads in a warp can
execute in lock-step without any explicit synchronization;
and 2) CUDA devices with computation capability of at
least version 3.X support intrinsics to perform data shuffling
operations across the threads within in a warp.

In addition to the basic scalar types, CUDA has explicit
SIMD vector types, for example, float2, float4. When
using thread local variables in vector types, the size of the
virtual SIMD vector increases by a factor of the size of the
vector type.

**Explicit SIMD vectors via shared memory** In order to
support both intra-vector and inter-vector SIMD operations
across SIMD lanes on the GPU, we also need explicit SIMD
vectors in memory. Without shuffle intrinsics, the only way
to communicate across GPU threads is through the memory.
According to the CUDA memory hierarchy, it is beneficial
to represent explicit SIMD vectors with shared memory.

Explicit SIMD vectors via shared memory can be used to
form super SIMD vectors with the sizes that are multiples
of a SIMD vector size. For example, if three explicit SIMD
vectors are allocated in shared memory contiguously, we can
treat these three SIMD vectors as a super SIMD vector with
the same starting address as the first SIMD vector. Super
SIMD vector plays an important role in optimizing non-unit
stride memory access and exploiting data locality.

### B. SIMD Operations

1) **Arithmetic Operations** CUDA devices bundle several
threads for execution. Each thread block is partitioned into
warps. The execution of warps are implemented by SIMD
hardware. The execution core of the processing units not
only can perform 32-bit integer and single- and double-
precision floating-point arithmetic operations, but also has
special function units (SFUs) to compute single-precision
approximations of log/exp, sin/cos, and rcp/rsqrt. Therefore,
it makes sense to implement SIMD arithmetic operations performing on the SIMD vectors organized in warps. Due to the lack of SIMD execution units in each streaming core, each SIMD operation on the data in SIMD types is decomposed into a sequence of scalar operations. Therefore, the granularity of a CUDA thread is coarsened by the vectorization when using SIMD data types.

2) Memory Operations: In this paper, we only consider unit and non-unit stride memory operations.

Unit stride loads/store Each SIMD lane of a SIMD vector is mapped to a thread in a warp. For a unit stride vector load/store, if it is aligned to 128 bytes, the memory access to the global memory would be coalesced.

Non-unit stride load/store Unlike CPUs, which support different data permutation instructions, GPUs have quite a limited data permutation capability. Data permutation operations on the GPU usually have to resort to memory, such as shared memory and global memory, to exchange data across threads with necessary synchronization.

In order to optimize non-unit stride loads and stores on GPU, we employ on-the-fly data layout transformation with the virtual SIMD vectors in shared memory. The on-the-fly data layout transformation works by first collecting a group of stride loads or stores with spatial locality. For the non-unit stride loads, a sequence of unit-stride vector loads with contiguous virtual SIMD vectors are generated to ensure all the data to be accessed by each stride load in the group is cached in shared memory in the form of virtual SIMD vectors. Then, the contiguous virtual SIMD vectors are converted into a super SIMD vector. Finally, each original stride memory access to the global memory is transformed into a stride access to the super SIMD vector. The transformation works similarly for the non-unit stride stores.

3) Data Reorganization Operations: In this section, we describe how to implement three widely used data reorganization operations.

Intra-vector Shuffle The intra-vector shuffle operation takes a SIMD vector and a mask vector as the input, and shuffles the data in the SIMD vector according to the mask vector. There are two options to generate mask vectors. For simple cases, we can adopt runtime generation at cost of control divergence due to the introduced if statements, as shown in line 4 - 9 of Fig. 6. However, sometimes these if statements can be optimized by predicated instructions. On the other hand, we can generate shuffle masks at compile-time and store the masks in global memory.

In order to ensure high performance, we always place immediate values in thread-local variables rather than shared memory. Therefore, when employing an intra-vector shuffle operation, all the values participating in the shuffle operation need to be cached into an explicit virtual SIMD register in shared memory. Then, the data shuffling operation is equivalent to random data access to shared memory.

The disadvantage of the earlier approach through an explicit virtual SIMD register is the cost of extra memory accesses to the shared memory. For devices with compute capability of at least 3.0, CUDA introduces a set of __shfl() intrinsics to permit exchanging of variables between threads within a warp without use of shared memory [9]. For the SwapEvenOddLanes operation, we can use the __shfl_xor(input, 0x1, 2), which partitions the SIMD lanes into sub-groups with the size of 2 and gives the source lane ID by a bitwise XOR.

Inter-vector Permutation The inter-vector permutation operation usually requires two vectors as the source vectors and a mask vector to specify the permutation pattern. Our solution to the inter-vector permutation operation is similar to the intra-vector shuffle operation via shared memory. We combine all the participating explicit virtual SIMD registers into a super-vector, and then replace the inter-vector permutation operation with an arbitrary memory access to the super-vector according to the permutation mask.

Blend The blend operation takes two vectors and a mask vector as the input, and selects a value for each element of the destination vector from either input vector according to the condition indicated by the mask vector. A simple solution is to convert a blend operation into a conditional assignment statement, which would be optimized by predicated instructions. Another solution is to convert the two input vectors into a super-vector in shared memory similar to the inter-vector permutation operation, and replace the blend operation with a memory gather operation on the super-vector.

C. Mapping Execution Model

In this section, we present three strategies of mapping SIMD lanes to GPU threads, direct mapping, flatten mapping and nested SIMD mapping, as shown in Fig. 7.

1) Direct Mapping: When a hyper loop is identified and the hyper loop parallelism is expressed as SIMD operations on SIMD vectors, it is intuitive to directly map each SIMD operation to a GPU thread as discussed in Sec. III-B. However, the width of SIMD lanes after the global SIMD lane optimization may differ from the width of SIMD units (the warp size). Given that we treat a warp of GPU threads as a SIMD computation unit, when mapping SIMD operations
with larger width than the warp size, we can partition the SIMD operations into several SIMD operations with width equal to the warp size. We call this mapping strategy direct mapping.

For instance, as shown in Fig. 7 (a), the SIMD operation to be mapped has a SIMD lane descriptor [0:64:1], indicating the width of this SIMD operation is twice larger than the warp size. Therefore, execution of this SIMD operation requires two warps of threads. In direct mapping, the SIMD operation is partitioned into two SIMD operations, and both of them are scheduled in the same warp and executed one after another.

2) Flatten Mapping: In contrast to direct mapping, another way of dealing with the SIMD operation with size larger than the warp size is to partition the SIMD operation into small SIMD operations and spread the resulting SIMD operations into different warps of GPU threads. As shown in Fig. 7 (b), the SIMD operation with a SIMD lane descriptor [0:64:1] is split into two SIMD operations with the SIMD lane descriptor [0:32:1]. We call this mapping strategy flatten mapping.

It is not always possible to apply flatten mapping due to the across SIMD lane operation. We consider the flatten mapping to be legal only when the following two conditions are satisfied: 1) the SIMD lane size of each node after global SIMD optimization is the same, and this uniform size is a multiple of the warp size; and 2) the data shuffling operations occurring in all the nodes could be split into several data shuffling operations, with size the same as the warp size. The second condition is to ensure that no matter which kind of across SIMD lane operation it is, the GPU threads engaged are never distributed into two different warps.

3) Nested SIMD Mapping: Both direct and flatten mapping may introduce data reorganization across GPU threads within a warp. Even with the hardware support for data shuffling within a warp without explicit shared memory access, the cost may be prohibitive. Using hyper loops to represent the inner-loop computation structure, the data reorganization in fact occurs only among the hyper loop iterations or within a hyper loop iteration when nested SIMD parallelism is discovered. Therefore, one way to get rid of the data shuffling operations across threads within a warp is to enlarge the the mapping granularity from a hyper loop iteration to a hyper loop. As shown in Fig. 7(c), each hyper loop instance with size of two is scheduled to a GPU thread. Because the main purpose of hyper loops is to expose an extra degree of parallelism for SIMD, we call this mapping strategy nested SIMD mapping.

Representing inner-loop computation structure as hyper loops could logically achieve AoS to SoA data layout transformation for the loops with data in AoS. When choosing the hyper loop as a schedule unit, it is of great importance to preserve the contiguous memory access exposed by hyper loops. Because contiguous accesses to global memory could be coalesced. In order to achieve contiguous memory access within a schedule unit, we need to use vectorized loads and stores through the CUDA vector types.

IV. IMPLEMENTATION

Our hyper loop parallelism vectorization for GPU is built on top of the Cetus source-to-source C compiler. The Cetus compiler is capable of auto-parallelization of C programs and generates C programs with OpenMP and Cetus directives [8]. The overall compilation flow of hyper loop parallelism vectorization for the GPU is illustrated in Fig. 8.

![Figure 8](image_url)
V. PERFORMANCE EVALUATION

A. Experiment Setup

1) Platforms: We evaluated our hyper loop parallelism vectorization for CUDA GPU on the following two platforms.

Jetson TK1 is a fully-featured embedded system designed for development of embedded and mobile applications. Jetson K1 incorporates a Tegra K1 that has a Kepler GPU with 192 cores, an Nvidia 4-plus-1 quad-core ARM Cortex-A15 CPU. The significant difference between Tegra K1 and other desktop GPUs is that the memory on Tegra K1 is physically unified but with separate CPU and GPU caches.

GeForce GTX 645 is a desktop GPU with the architecture of Kepler GK106. It supports computation capability 3.0 and has 576 CUDA cores and 1024MB GDDR5 memory.

B. Test-cases

We chose a range of parallel loops featuring both unit and non-unit strides as listed in Table I to demonstrate the performance of our proposed technique. For non-unit stride tests, we consider two representative strides: 2 and 3. For other strides, our technique can work similarly.

Table I

<table>
<thead>
<tr>
<th>Stride</th>
<th>Test-case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Array Copy</td>
<td>Copy 1D array.</td>
</tr>
<tr>
<td></td>
<td>1D Dot</td>
<td>Dot operation on 1D vectors.</td>
</tr>
<tr>
<td></td>
<td>Blur</td>
<td>Blur in image processing on a 1D linearized array.</td>
</tr>
<tr>
<td>2</td>
<td>C-Saxpy</td>
<td>Complex vector operation.</td>
</tr>
<tr>
<td>3</td>
<td>Vec-Copy</td>
<td>Basic operations on 3D vectors in AoS in image processing</td>
</tr>
<tr>
<td></td>
<td>Vec-Dot</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vec-Crossproduct</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vec-Rotation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vec-Normalization</td>
<td></td>
</tr>
</tbody>
</table>

C. Performance Evaluation and Analysis

We compare the performance of HLP vectorization for CUDA GPU against the direct coarse-grain loop parallelism mapping. When generating the CUDA code for a parallel loop nest, the most common mapping strategy adopted is to directly map a loop iteration to a CUDA thread. The baseline CUDA code for performance comparison is generated by OpenMPC [8] with the direct coarse-grain loop parallelism mapping.

1) Unit Stride Tests: Array copy is a trivial test-case to demonstrate whether our nested SIMD mapping could yield any performance improvement. As shown in Fig. 10, vectorized loads/stores using CUDA built-in vector type float2/float4 would not give any performance gain on the Jetson TK1 platform while a speedup of up to 1.1 × on GeForce GTX 645. We suspect that the vectorized loads and stores are not well supported on the Jetson TK1 platform.

Although performing only vectorized loads and stores improves performance on GeForce GTX 645, when computation is involved, the performance gain is reduced by the long latency of the vectorized load instructions. For example, LD.E.64 can load 2 floats into two registers, and all the instructions using these two registers explicitly have to wait until the load is finished. The 1D dot has no other computations to overlap the memory latency; thus, regardless of the kind of mapping, the performance almost stays the same.

Compared to 1D dot, which has no temporal locality, blur has lots because of the stencil computation. Temporal locality is often exploited to handle unaligned vectorized memory operations in vectorization for CPUs [10]. Similarly, we could exploit temporal locality to optimize memory access to global memory. When the nested SIMD mapping is applied, a group of threads that share some data are squeezed into a single thread. If there is shared data between the threads being coarsened, coarsening them will reduce the number of memory operations to global memory. As shown in Fig. 10, nested SIMD mapping improves the performance of blur by 1.4×.

2) Non-unit Stride Tests: Stride 2 - C-saxpy C-saxpy is a good candidate to evaluate different execution mapping strategies on the CUDA GPU. As shown in Fig. 11, all
the mapping strategies proposed in this paper yield better performance, when compared to the direct coarse-grain loop parallelism mapping, which is common in existing compilers with semi-/fully automatic code generation for CUDA. Speedups up to $1.7 \times$ are obtained from HLP vectorization. The major performance contribution comes from the global memory load efficiency that is boosted from 50% to 100%. The performance difference between the flatten mapping and direct mapping demonstrates that instruction level parallelism is important for performance as well. The flatten mapping splits the SIMD operations with width 64 into two short operations with width 32, and maps the resulting operations onto two warps. Consequently, the ILP in each warp is reduced.

Similar to the 1D dot, nested SIMD mapping does not give extra performance gain for the csaxpy over the direct mapping. The nested SIMD mapping with \texttt{float2} and \texttt{float4} coarsens the thread granularity of the direct mapping by a factor of 2 and 4, respectively. The thread coarsening reduces the overall number of threads. But the coalesced global memory access via vectorized loads and stores and the extra ILP by threading coarsening make the performance on a par with the direct mapping. However, csaxpy is a memory bounded kernel so that the performance difference between \texttt{float2} and \texttt{float4} is slight.

**Stride 3** As shown in Fig. 12 and Fig. 13, HLP vectorization can achieve significant speedups for the stride-3 test-cases on both platforms. The profiling data obtained by the \texttt{nvprof} in the CUDA toolkit shows that the global memory load efficiency is improved to 100% from 33.3%. For \texttt{Vector Copy}, all the stride memory accesses to global memory from the direct mapping are coalesced thanks to the hyper loop structure. Other stride-3 test-cases require on-the-fly data layout transformation via shared memory for load and store operations. The great speedups achieved — up to a factor of 2.5 — demonstrate that HLP vectorization can be an effective way to improve the memory performance on the CUDA GPU.

**VI. RELATED WORK**

There is extensive work on optimizing memory performance for the GPGPUs. For example, Jang et al. [11] introduced vectorization via data transformation to benefit vector-based architectures (e.g. AMD GPUs) and algorithmic memory selection for scalar-based architectures (e.g. Nvidia GPUs). Che et al. [12] proposes a simple API to allow programmers to optimize memory mappings to improve the efficiency of memory accesses on heterogeneous platforms. Xu et al. [13] introduced a set of compiler directives to help the programmer to apply global data layout transformations for better vectorization. Instead of global data layout transformation, our proposed technique optimizes the memory performance by exploiting the computation structures of the vectorizable loops and applying on-the-fly data layout transformation.

Employing vectorization for the GPU is not a new idea. Kerr et al. [14] put forward a dynamic compiler to compile explicitly data-parallel kernels for SIMD functional units. In contrast, our work focuses on how to exploit SIMD parallelism when automatically extracting data-parallel kernels and addressing the problem of non-unit stride memory access with our vectorization technique. Automatic code generation from high-level languages
to CUDA has been studied since CUDA was put forward. One pioneering approach is hiCUDA [15], a high-level directive-based language for CUDA. Other directive based programming methods for heterogeneous devices have evolved into programming standards, such as OpenMP 4.0 [1] and OpenACC [2]. Since these standards have been established, many attempts have been made to explore and evaluate the ways of mapping coarse-grain loop parallelism to the heterogeneous devices, including both the commercial compilers, such as PGI OpenACC compiler, and open-source compilers such as OpenARC [8], OpenUH-ACC [16]. However, existing compiler work for directive based programming methods concentrates only on how to map the execution model of the programming model to the devices. For example, Tian et al. [16] discussed on how to map the gang, worker, vector execution model of OpenACC to the CUDA execution model. Even with explicit annotations for SIMD parallelism, the compiler is not able to employ suitable vectorization and efficiently map the vectorized code to the heterogeneous devices. To the best of our knowledge, our work is the first work on employing a variant of super-word level parallelism vectorization [4] [5] [17], HLP vectorization, in automatic C-to-CUDA.

VII. Conclusion

In this paper, we put forward a compiler framework to extract hyper loop parallelism in vectorization and map the parallelism efficiently on the CUDA GPU. Our method achieves thread coarsening, which can reduce memory operations in the presence of data locality, and optimizes uncoalesced memory access to global memory. In addition, the introduction of hyper loop parallelism further refines the mapping granularity between coarse-grain loop parallelism and GPU threads. Our vectorization techniques are general, and could be adopted in existing directive based programming models for GPUs to improve the memory performance.

Our experimental evaluation demonstrates that our proposed approach can deliver speedups of up to 2.5 \times compared to the direct coarse-grain loop parallelism mapping.

REFERENCES


