Optimizing GPU Register Usage: Extensions to OpenACC and Compiler Optimizations

Xiaonan Tian*, Dounia Khaldi†, Deepak Eachempati*, Rengan Xu* and Barbara Chapman*†
*Dept. of Computer Science
University of Houston
Houston, TX
Email:{xtian2, dreachempati, rxu6, bchapman}@uh.edu
†Institute for Advanced Computational Science
Stony Brook University
Stony Brook, NY
Email:{dounia.khaldi, barbara.chapman}@stonybrook.edu

***Abstract*—Using compiler directives to program accelerator-based systems through APIs such as OpenACC or OpenMP has increasingly gained popularity due to the portability and productivity advantages it offers. However, when comparing the performance typically achieved to what lower-level programming interfaces such as CUDA or OpenCL provides, directive-based approaches may entail a significant performance penalty. To support *massively parallel* computations, accelerators such as GPGPUs offer an expansive set of registers, larger than even the L1 cache, to hold the temporary state of each thread. Scalar variables are the mostly likely candidates to be assigned to these registers by the compiler. Hence, scalar replacement is a key enabling optimization for effectively improving the utilization of register files on accelerator devices and thereby substantially reducing the cost of memory operations. However, the aggressive application of scalar replacement may require a large number of registers, limiting the application of this technique unless mitigating approaches such as those described in this paper are taken.**

In this paper, we propose solutions to optimize the register usage within offloaded computations using OpenACC directives. We first present a compiler optimization called *SAFARA* that extends the classical scalar replacement algorithm to improve register file utilization on GPUs. Moreover, we extend the OpenACC interface by providing new clauses, namely `dim` and `small`, that will reduce the number of scalars to replace. SAFARA prioritizes the most beneficial data for allocation in registers based on frequency of use and also memory access latency. It also uses a static feedback strategy to retrieve low-level register information in order to guide the compiler in carrying out the scalar replacement transformation. Then, the new clauses we propose will extremely reduce the number of scalars, eliminating the need for more registers.

We evaluate SAFARA and the new clauses using SPEC and NAS OpenACC benchmarks; our results suggest that these approaches will be effective for improving overall performance of code executing on GPUs. We got up to 2.5 speedup running NAS and 2.08 speedup while running SPEC benchmarks.

***Index Terms*—OpenACC; GPUs, Scalar Replacement; Register Usage Optimization; Static Feedback**

I. Introduction

The hardware architectures of CPUs and GPUs are very different because they target different types of applications. CPUs are general-purpose processors and optimized to achieve high performance on sequential code. GPUs were originally designed for graphical computing, particularly SIMD operations. As GPUs are increasingly programmable, they become massively parallel architectures. The memory system is one of the most significant differences between CPUs and GPUs. GPUs typically offer high-bandwidth memory to feed a large number of parallel threads and use threads scheduling to overlap computations with long-latency memory access.

GPUs platforms are a natural target for achieving data parallelism by parallelizing loops and arrays, where each iteration of the loop is executed on different data chunks of an array. The characteristics of the memory access patterns can have a huge impact on the overall performance. However, analyzing these patterns is extremely challenging in low-level programming models targeting GPUs, such as OpenCL and CUDA, due to the frequent presence of pointer operations. Data resides in the global memory and is passed by pointer to kernel parameters. Extracting useful reuse information based on the analysis of array indices poses significant difficulties. On the other hand, high-level directive-based APIs for programming GPUs, such as OpenACC [11] and OpenMP [12], preserve high-level language features that can allow a compiler to apply classical analysis and optimizations and generate high-quality GPU code. Using directives that preserve the usage of arrays also allows the compiler to retain reuse information, array dimensions, etc. For these reasons, in this work we consider OpenACC programs for demonstrating our proposed optimization scheme. Note that the same strategy can also be used for OpenMP programs utilizing directives for accelerators, available in OpenMP 4.x.

Scalar replacement is a classical compiler optimization that can be used to eliminate frequent, redundant memory accesses of data in a computational region (a contiguous code fragment). Scalar replacement converts array references to scalar variable, and then assigns these scalar variables to registers. By replacing repeatedly used array references, the compiler can reduce expensive memory load/store operations and instead make the data readily available for reuse in the register files, before storing them back in memory when not needed anymore. The classical scalar replacement algorithm [10] works well for sequential programs running on traditional CPU architectures. However, the algorithm does not work well on parallel systems such as GPUs, which contain complex memory hierarchies.

In this paper, we present SAFARA, an iterative GPU-aware scalar replacement algorithm we developed for fully exploiting the rich set of register file resources that are typical of GPU architectures. SAFARA takes advantage of data reuse and memory latency when allocating to registers. Moreover, since the aggressive application of scalar replacement increases register pressure, which may lead to low-thread occupancy or cause register spilling, and thus hurt performance, we propose two extension clauses to OpenACC to reduce the register usage. The idea behind these two clauses, namely `dim` and `small` is to save some register files for scalar replacement by removing unnecessary array offset computations.

This paper makes the following contributions.

- We introduce a new algorithm to assist register allocation called SAFARA, based on feedback information regarding register utilization and a memory-latency-based cost model to select which array references should be replaced by scalar references.
- Two new clauses are proposed (`dim` and `small`) to describe array dimensions and size information. The compiler then can use these information to eliminate unnecessary variables needed in offset computation, and thus reduce register usage.
- We implemented SAFARA and the support for the new clauses in the OpenUH compiler and evaluated it using the NAS and SPEC benchmarks on NVIDIA GPU. We got up to 2.5 speedup running NAS and 2.08 speedup for running SPEC benchmarks.

The organization of this paper is as follows. Section II provides an overview of the NVIDIA GPUs architecture, the OpenACC programming model, and the OpenACC implementation within the OpenUH compiler. Section III introduces our new scalar replacement algorithm in OpenACC offload regions called SAFARA. Section IV presents the two new clauses we suggest to add to the OpenACC API in order to optimize register usage. Performance results are discussed in Section V. Section VI highlights the related work in this area. Finally, we conclude our work in Section VII.

II. Background

In this section, we provide a brief introduction about the architecture of GPUs, their memory hierarchy, OpenACC and its implementation in the OpenUH compiler.

A. GPU Architecture

Modern GPUs consist of multiple streaming multiprocessors (SMs or SMXs); each SM consists of many scalar processors (SPs, also referred to as cores). Each GPU supports the concurrent executions of hundreds to thousands of threads following the Single Program Multiple Data (SPMD) programming model, and each thread is executed by a scalar core. The smallest scheduling and execution unit is called a warp, which is composed of 32 threads. Warps of threads are grouped together into a thread *block*, and blocks are grouped into a *grid*. Both thread block and grid can be organized into a one-, two- or three-dimensional topology.

B. GPU Memory Hierarchy

Modern GPUs deploy a deep memory hierarchy which includes several different memory spaces. Each of them has special properties. For example, accesses to global memory could get coalesced/uncoalesced, accesses to texture memory could come with spatial locality penalty, accesses to shared memory could suffer from bank conflicts and accesses to constant memory could be broadcast. All of them are organized into a complex and deep GPU memory hierarchy which is extremely different from the CPU memory model. Note that memory coalescing is one of the key data locality features that is provided by modern GPU architectures to exploit this locality within a warp. The memory request is sent out by each thread, and memory transactions from the same warp are grouped together into large transactions. Basically, coalesced memory transaction means that consecutive threads will access consecutive memory addresses.

Figure 1 shows an overview of the memory hierarchy in NVIDIA Kepler GPUs. Kepler GPUs have a global memory space that is accessible by all threads in a grid, and this is the space that the CPU memory can communicate with. Shared memory is allocated per thread block, and is accessible by threads within the thread block. Because the shared memory is on-chip, latency is much lower than for global memory. The L1 cache in the NVIDIA Kepler architecture is reserved only for local memory accesses by default, such as register spilling and stack data. Global loads/stores are cached in L2 only. Read-only Data Cache was introduced in the latest NVIDIA Kepler architecture. Each SMX has a 48KB Read-only Data Cache. Each SP in SMX accesses data via the read-only cache when the data is read-only for the lifetime of the CUDA kernel. Each SMX has 256KB register files and each thread can use up to 255 32bit register files.

C. OpenACC Programming Model

OpenACC is a high-level programming model that can be used to port HPC applications to different types of accelerators such as NVIDIA GPUs, AMD GPU & APU, and Intel Xeon Phi. It provides directives, runtime routines and environment variables as its programming interfaces. The execution model assumes that the main program runs on the host, while the compute-intensive regions of the program are offloaded to the attached accelerator. The accelerator and the host may have

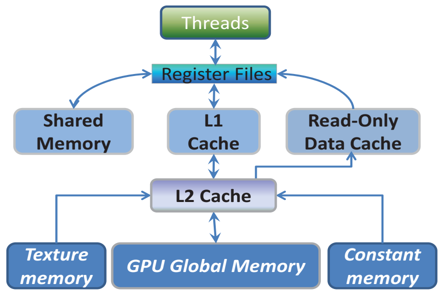

Fig. 1: NVIDIA Kepler GPU Memory Hierarchy

separate memories, and the data movements between them may be controlled explicitly. OpenACC provides a rich set of data transfer directives, clauses and runtime calls as part of its standard. To minimize the performance degradation due to data transfer latency, OpenACC also allows asynchronous data transfers and asynchronous computations with the CPU code, thus enabling overlapping data movement and computation.

OpenACC uses the `parallel` or `kernels` constructs to define a compute region that will be executed in parallel on the device. The `loop` construct is used to specify the distribution of iterations. The purpose of having both `parallel` and `kernels` is that the `parallel` construct provides more control to the user while the `kernels` one offers more control to the compiler. The `reduction` clause is allowed on a loop construct. We denote both parallel and kernels regions as 'offload' regions in this paper. The execution model of OpenACC assumes that the main program runs on the host, while the compute-intensive regions of the main program are offloaded to the attached accelerator. In the memory model, usually the accelerator and the host CPU use separate memory addresses to prevent conflicts between CPU and accelerators. OpenACC 1.0 discusses different types of data transfer clauses. Some additional data directives and runtime routines to control the unstructured data lifetime have been added in the 2.0 specification.

D. OpenACC Implementation in OpenUH

The OpenACC support that has been implemented in OpenUH [17], [16] addresses three areas: (1) an extended front-end that accepts the OpenACC directive syntax in C and Fortran, (2) middle-end/back-end analyses, optimizations and translations for OpenACC offload regions, and (3) enhanced IR-to-source tools for supporting CUDA/OpenCL kernel function translation. As shown in Figure 2, we use a source-to-source translation technique to translate OpenACC offload regions into CUDA/OpenCL code. NVCC is the CUDA compiler. OpenUH directly generates object code for the x86 host CPU.

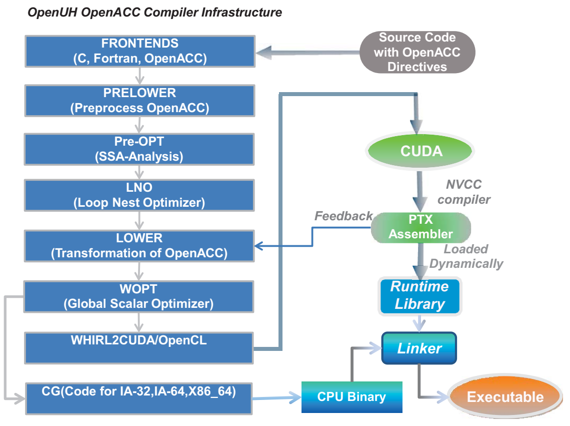
OpenUH OpenACC Compiler Infrastructure
FRONTENDS (C, Fortran, OpenACC)
Source Code with OpenACC Directives
PRELOWER (Preprocess OpenACC)
Pre-OPT (SSA-Analysis)
CUDA
LNO (Loop Nest Optimizer)
NVCC compiler
Feedback
PTX Assembler
LOWER (Transformation of OpenACC)
Loaded Dynamically
WOPT (Global Scalar Optimizer)
Runtime Library
WHIRL2CUDA/OpenCL
Linker
CG(Code for IA-32,IA-64,X86_64)
CPU Binary
Executable

Fig. 2: OpenUH OpenACC Compiler Infrastructure

III. SAFARA: STATIC FEEDBACK-BASED REGISTER ALLOCATION ASSISTANT FOR GPUS

Scalar replacement (SR) is a classical optimization that can be applied to improve utilization of register files. In this section we present what we view as limitations of the state-of-the-art algorithm, by Carr and Kennedy [1], and we introduce our new algorithm called SAFARA.

A. The Carr-Kennedy Algorithm

The scalar replacement algorithm [1] includes three phases: (1) a dependence distance-based data reuse analysis, (2) a moderation model of register pressure, and (3) the scalar replacement transformation. The Carr-Kennedy algorithm [1] uses input and flow dependence analysis to find the array references. If all of the reused memory references found in the first phase are replaced with scalar, the performance of the application may slow down because of register spilling. The moderation of register pressure is used to find out the most beneficial memory references that can be replaced with scalars. Once the memory references that are chosen to be replaced with scalars are determined, the compiler performs the scalar replacement transformation to replace the memory references with scalars. However, the Carr-Kennedy algorithm cannot be directly applied to OpenACC offload regions. In this section, we present its limitations when applied to GPUs.

1) Creation of Inter-iteration Dependences in Parallelized Loops: The first limitation of the Carr-Kennedy algorithm is that it may translate an independent loop into a dependent loop that cannot be parallelized. An example of a parallel loop is provided in Figure 3, where the array references `b[i]` and `b[i+1]` introduce an input data dependence edge which has a dependence distance of 1. Therefore, the data loaded in `b[i+1]` at iteration i will be used in array reference `b[i]` at iteration i+1. The Carr-Kennedy algorithm will detect the data reuse opportunities and perform the scalar replacement optimization. The loop will be thus transformed into the code shown in Figure 4, which has only 1 array reference in the loop body. The loop in Figure 4 introduces loop-carried flow dependences across iterations between `b1` and `b[i+1]`. Consequently, the loop cannot be parallelized. This conflicts with the goal of OpenACC which is to expose parallelism to be exploited by the massively parallel accelerator. In fact, executing the loop sequentially by each thread of the GPU will lead to a significant performance penalty. Therefore, in our solution, we will prevent scalar replacement from being performed across iterations, if the loop can be parallelized.

```
for(i=1; i<=SIZE ; i ++)
        a[i] = (b[i] + b[i+1])/2;
```

Fig. 3: Before SR: iterations are independent

```
b1=b[1]
for(i=1; i<=SIZE ; i ++){
        b2=b[i+1];
        a[i] = (b1 + b2)/2;
        b1 = b2;
}
```

Fig. 4: After SR: iterations are dependent

```
#pragma acc loop gang vector
for(j=1; j<=JSIZE ; j ++){
    c[j] = b[j][0] + b[j][1];
    d[j] = c[j] *b [j][0];
#pragma acc loop seq
    for(i=1; i<=ISIZE ; i ++){
        a[i][j] += a[i-1][j] + b[j][i-1] +
             a[i+1][j] + b[j][i+1];
    }
}
```

Fig. 5: Sample OpenACC program before SR

2) Cost Model not Adapted to GPUs: The memory access latency in GPUs is different from the one of traditional CPU systems. In the Carr-Kennedy algorithm, the metric used is how many memory accesses can be removed. For this, a model of register pressure moderation is designed to select the most beneficial references to be transformed into scalar variables if limited register files are available. For instance, in Figure 5, references to each array `a` and `b` require 3 temporary variables. In the Carr-Kennedy algorithm, when the number of available registers is limited, the array references of `a` have higher priority to be replaced with scalar variables because it is used one more time than `b`. However, in GPUs, another metric should be taken into account due to the memory hierarchy of GPUs; this is the second limitation of this algorithm for our purpose. In fact, since iterations in Loop `j` are distributed across the x-dimension of each thread in each thread block, the memory access in `a` are coalesced within a warp. Meanwhile, the memory accesses in `b` are uncoalesced within a warp. Thus, the memory access latency of `b` is much higher than that of `a`. In this case, replacing array references of `b` will have a better benefit than replacing the references of `a`.

B. SAFARA: StAtic Feedback-bAsed Register allocation Assistant for GPUs

SAFARA addresses the two limitations presented in the previous section. For the 1st limitation, the scalar replacement approach can be divided into intra-iteration and inter-iteration transformations. If the loop is identified as parallelized in OpenACC, only the intra-iteration SR is performed to avoid to sequentialize it. Otherwise, if the loop is sequential, then inter-iteration SR can be safely applied. As for the 2nd limitation, three new components are integrated.

1) First, during the dependence analysis to retrieve data reuse, we count how many times every array reference is used (read/write). Then, array references are classified into four categories according to the memory hierarchy in the GPU: shared, constant, read-only (available in NVIDIA Kepler GPUs only) and global memory access [1]. Read-only and global memory data accesses can be further divided into coalesced and uncoalesced accesses. Each of them has different memory access latency [6].
2) Second, we use GPU tools to pinpoint the register usage information and then feed it back to the OpenACC compiler to perform the SR. The NVIDIA GPU tool we used is called *PTXAS Info*.
3) Third, using the information from the first step, we estimate the cost of each array reference R belonging to a memory space M, using the formula *reference_count(R)* $\times$ *memory_access_latency(*M*)*. Then, all array references can be sorted from higher to lower cost. After that, we select the most beneficial memory references to be replaced by scalar variables.
4) Go to Step 2 until all the registers are used or all the reused references are replaced.

In the following subsections, we explain in details the methodology followed by these steps.

1) Array Reference Analysis in SAFARA: Memory access pattern analysis is introduced into SAFARA to classify different memory access modes. Basically, the memory access latency depends on where the data is located and how the data is accessed. There are several different memory spaces in modern GPU architectures. For the NVIDIA Kepler GPUs, there are shared memory, read-only global data, read/write global data, constant memory and texture memory. Read-only data can be placed in the global memory and cached by the read-only data cache in each SM. While building the dependence graph, the compiler performs array index analysis to determine if the memory access is coalesced or not. The index analysis that is used in our algorithm is inspired from [8] which proposes a mathematical model that captures and characterizes memory access patterns inside nested loops.

[1]Note that in our implementation, we only consider read-only and global memory accesses.

This is used to recognize if the memory access is coalesced or not.

2) Iterative Register Information Feedback to SAFARA: We use register utilization information from GPU tools to improve the scalar replacement transformation done in SAFARA. In traditional CPU compilers that perform register allocation and directly generate actual assembly code, this register information is available during compile time. However, since GPU architectures change dramatically between generations, compilers for GPUs generate a stable, virtual ISA that spans multiple GPU generations and use pseudo registers. For example, NVIDIA uses "PTX". There are unlimited pseudo register numbers available in the virtual ISA. The compiler cannot determine how many hardware registers have been used. Vendors, including NVIDIA, provide closed-source low-level assembler tools to translate the virtual ISA into actual GPU assembly code and allocate hardware registers.

In our work, we propose to assist the compiler by using feedback information from these tools to calculate how many hardware registers are available. Moreover, backend compilation is performed multiple times. The first time does not perform any scalar replacement; it is only dedicated to invoking the GPU assembler tool to output the hardware register usage information. The following compilation combines the register usage information and register upper limit specified by the hardware limit (for instance the maximum number that can be used in NVIDIA Kepler GPU is 255 registers per thread) to determine the availability of registers. If there are available registers, the scalar replacement optimization is invoked. The compiler analysis lists all the memory references that can satisfy the scalar replacement requirements. If the number of candidates is less than the available register count, all of them are replaced by scalars. Otherwise, the cost model based on array references selection is invoked.

3) Cost Model for Array References Selection: In contrast to traditional CPU architectures, overuse of GPU register files causes severe performance degradation due to register spilling as well as lowering of thread concurrency. This raises the issue of how to select good memory references if their number is larger than the number of available registers. We use a cost model to prioritize memory accesses for replacement. It is based on the memory access latency, which is used to estimate the potential cost of different memory accesses. The model consists of two factors: memory access latency L and references count C. The potential access cost is computed as $L \times C$. Note that the method used to measure the latency of GPU memory accesses employs the microbenchmark proposed by [19].

4) Running Example with SAFARA: In the following, we demonstrate the application of SAFARA on the example shown in Figure 5. After the first time we apply the first iteration of SAFARA, we suppose that the GPU tool outputs 26 as the number of registers that are used. We suppose that the hardware limit is only 30 registers. So, the number of available registers found by our algorithm is 4. In the second iteration of SAFARA, the scalar replacement is applied on Array `b` using 3 registers (recall that Array `a` is coalesced and should not be put in registers) and the code will be transformed into the code shown in Figure 6. Note that this code will need 4 iterations to complete. After four iterations of the feedback feature of SAFARA, we saturate the number of registers available and we replace the most beneficial memory accesses.

```
#pragma acc loop gang vector
for(j=1; j<=JSIZE ; j ++){
    c[i] = b[j][0] + b[j][1];
    d[j] = c[j] *b [j][0];
    b0=b[j][0];
    b1=b[j][1];
    for(i=1; i<=ISIZE ; i ++){
        b2 = b[j][i+1];
        a[i][j] = a[i-1][j] + b0
                + a[i+1][j] + b2;
        b0 = b1;
        b1 = b2;
    }
}
```

Fig. 6: Sample OpenACC program after SAFARA

IV. Proposed Extensions to openACC: `dim` and `small` New Clauses

Scalar replacement is a classical memory optimization algorithm to reduce redundant memory access. In the previous section III, we presented an extension to SR by providing different techniques and we called this extension SAFARA. However, the aggressive application of scalar replacement increases register pressure, which may lead to low threads occupancy or cause register spilling, and thus hurt performance. To confirm this result, we perform a study on the SPEC benchmark suite and we show experimental results in Figure 7. The experimental setup is provided in SectionV-A. In this study, we found that SAFARA provides either very small performance improvement or sometimes slows down the application when registers are exhaustively used by threads, because this leads to low threads occupancy; we also found that no register spilling happened based on SAFARA feedback information. Finding the best combination between what is the optimal number of registers to use by each thread and thread occupancy is a complex problem [18]. Note that this paper does not solve this problem of finding the best number of registers to get peak performance. However, this work focuses on minimizing the number of registers. In this section, we propose a solution at the API level of OpenACC to reduce the number of scalars that will be held potentially in registers. This will save some registers to use by each thread and thus increase threads occupancy.

Array references are frequently used in high-level programming models like OpenMP and OpenACC that are API extensions to C/C++ and Fortran languages. SPEC benchmarks for example contain a lot of array references. When the offload region is translated into the GPU lower level kernel routines (using CUDA for example), the array reference is

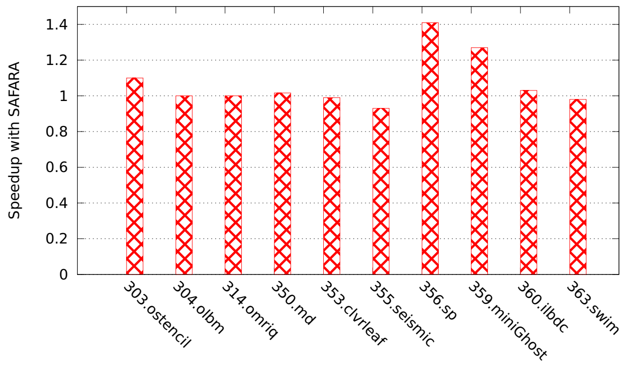

Fig. 7: Speedup results of SPEC benchmark suite with SAFARA

represented using a pointer and offset calculation operations. The idea behind the two clauses we want to introduce is to save some register files for scalar replacement by removing unnecessary array offset computations. The programmer can directly use these two clauses to pass array information to the compiler. We provide a motivation example in Figure 8 which shows a snippet of one of the offload regions in SPEC Accelerator 355.seismic benchmark. In this code, Loop j iterations are distributed across the y-dimensional threads and Loop i iterations are carried by x-dimensional threads in the GPU. The iterations in Loop i and Loop j are evenly distributed across all the threads and each thread only executes one iteration. The innermost Loop k is executed sequentially and SAFARA can be applied over array references across the k iteration.

```
!$acc kernels loop gang(NY/2) vector(2)
do j = 2,ny
  !$acc loop gang((NX-1+63)/64) vector(64)
  do i = 1,nx-1
    !$acc loop seq
    do k=2,nz
      ...
      value_dz = (vz_1(i,j,k)-vz_1(i,j,k-1))/h &
               + (vz_2(i,j,k)-vz_2(i,j,k-1))/h &
               + (vz_3(i,j,k)-vz_3(i,j,k-1))/h
      ...
    enddo
  enddo
enddo
```

Fig. 8: Snippet code from SPEC 355.seismic benchmark

A. *dim* Clause

In multiple scientific kernels including SPEC, the arrays are allocatable arrays (case of Fortran) or Variable-Length Arrays (VLA) (case of C/C++). These arrays are dynamically allocated; the dimensional information of the array is held in a dope vector data object generated by the compiler. In the example provided in Figure 8, the offset calculation requires five additional compiler-generated temporary variables to hold the lower bound and length for each dimension in Fortran, while in VLAs in C/C++, we need temporary variables to hold the length for each dimension, since the lower bound is always zero. The listing below details the computation of the offsets and reference addresses for the three array references in Figure 8. The variables t0,...,t14 hold dimensional information for each array. Note that 15 scalar variables are used to keep the boundary information and calculate the offsets of the three arrays.

```
offset0 = (i-t0) + t3 * ((j-t1) + t4 * (k-t2))
vz_1(i,j,k) --> *(vz_1 + offset0)

offset1 = (i-t5) + t8 * ((j-t6) + t9 * (k-t7))
vz_2(i,j,k) --> *(vz_2 + offset1)

offset2 = (i-t10) + t13*((j-t11)+t14*(k-t12))
vz_3(i,j,k) -->  *(vz_3 + offset2)
```

However, these three arrays have exactly the same dimensions. If the compiler has this equality of dimensions information, the array references address computation can be optimized into a simplified version which is shown in the listing below:

```
offset0 = (i-t0) + t3 * ((j-t1) + t4 * (k-t2))
vz_1(i,j,k) --> *(vz_1 + offset0)
vz_2(i,j,k) --> *(vz_2 + offset0)
vz_3(i,j,k) --> *(vz_3 + offset0)
```

At the compilation time, the compiler has no idea whether these arrays have the same dimension. Therefore, we propose a new clause `dim` to be added to `kernels` and `parallel` directives to specify which arrays share the same dimension(s). At the GPU code generation phase, the compiler can take advantage of this clause information and optimize the offset computation. Note that, in this specific example, the number of registers needed can be reduced to 5, which corresponds to (number of scalars)/(number of arrays). The `dim` clause syntax is shown below:

```
Fortran:
!$acc kernels/parallel &
   dim([(lb1:len1,...,lbN:lenN)](A1,...,),...)

C/C++:
#pragma acc kernels/parallel \
   dim([len1]...[lenN](A1,...,),...)
```

Note that dimension data in the clause syntax is optional. If the user does not specify the dimension data, as follows, the compiler can automatically load lower bounds and length data from one of the array's dope structure:

```
!$acc kernels dim( (vz_1, vz_2, vz_3))
```

However, we recommend providing complete information (dimensions and arrays) because the compiler can simplify further the offset computation, in particular when the lower bound is zero, as below:

```
!$acc kernels &
dim((0:NX, 0:NY, 0:NZ)(vz_1, vz_2, vz_3))
```

B. `small` Clause

In the case of 64-bit scalar, the register allocation phase in the compiler replaces it into a 64-bit register. However, in the GPU, the general purpose registers are only 32bits. So, a 64-bit scalar requires two consecutive GPU registers to hold the entire value of the scalar.

On 64-bit machines, the compiler uses a pointer type of 64-bits size while an offset is also a 64-bit integer. However, if the array size is less than 4GB, array references address computation can be represented with 64-bit addresses and 32-bit integer offsets. The size of register files used for offset computations can thus be reduced by up to half. In fact, small array sizes are common in current applications due to the limited device memory. Note that when the array is a static array in both C or Fortran, the compiler can detect the array size and decide whether 32-bit integers are enough to handle the offset value computation. However, when an allocatable array or VLA is used, the compiler cannot figure out the array size. By default, the compiler will use 64-bit integer to be safe.

Therefore, we propose the new clause *small* to tell the compiler that the offset of an array can be represented within a 32-bit integer. Here, a small array means the array size is smaller than 4GB and array references of such arrays can be represented with an array address plus a 32-bit integer offset. The `small` clause syntax is shown below:

```
Fortran:
!$acc kernels/parallel &
   small(A1,...,An)

C/C++:
#pragma acc kernels/parallel \
   small(A1,...,An)
```

If we apply this clause to the previous example, as follows, the register number can be divided by 2:

```
!$acc kernels &
dim((0:NX, 0:NY, 0:NZ)(vz_1, vz_2, vz_3)) &
small(vz_1, vz_2, vz_3)
```

Note that in the case where the user provides incorrect information inside the proposed clauses, the compiler can generate two versions of each kernel: (1) optimized kernel: that assumes that the information from the user is correct, (2) unoptimized kernel: that ignores the clauses. Also, the compiler can generate a segment of code responsible for verifying the correctness of the clauses. At runtime, this segment will be run and a decision will be made to execute the optimized or unoptimized kernel.

V. Evaluation

To assess the ability of SAFARA augmented with the proposed clauses to optimize the register usage in GPUs, we implemented SAFARA and the `dim` and `small` clauses in the OpenUH compiler. This section discusses experimental results for our implementation on two sets of benchmarks: (1) SPEC ACCEL suite [9] and (2) NPB OpenACC suite [20].

A. Experimental Setup

We gathered performance results via experiments we performed on a NVIDIA GPU. The machine we used includes a host with 8 cores Intel Xeon x86_64 CPU and 32GB main memory; the attached GPU is a K20Xm with 5GB global memory. CUDA 6.5 is used for the OpenUH backend GPU code compilation with ”-O3” optimization. For the comparative analysis, we used one of the major commercial OpenACC compiler vendors, namely PGI V15.9. We use ”-O3,-acc -ta=nvidia,cc35” for the PGI compiler options. To obtain reliable results, all experiments were performed five times and then the average performance was computed.

B. SPEC and NPB OpenACC Benchmarks

We chose these benchmarks since their codes are large and complex enough to simulate the behavior of real applications. The SPEC ACCEL benchmark suite includes two independent suites which are OpenCL and OpenACC ones. The SPEC OpenACC suite includes both C and Fortran applications and is used in this work. NPB OpenACC benchmarks are written in C language. NPB are well recognized for evaluating current and emerging multi-core/many-core hardware architectures, characterizing parallel programming models and testing compiler implementations. NPB OpenACC suite offers open-source benchmarks. In this benchmark suite, we used six benchmarks: EP (Embarrassingly Parallel), CG (Conjugate Gradient), MG (MultiGrid), SP (Scalar Pentadiagonal), LU (Lower-Upper symmetric Gauss-Seidel), BT (Block Tridiagonal). Note that the problem size ”C” is used for evaluation for all the six benchmarks.

C. Performance Evaluation

Figure 9 shows speedup results for the SPEC ACCEL benchmark suite after applying first the `dim` and `small` clauses to reduce the number of required registers and then SAFARA to make the scalar replacement. Note that speedup numbers correspond to the optimizations applied one after an another (`small`, then `small` + `dim`, then `small` + `dim` + SAFARA). Note that Benchmarks 303, 304, 314 are C benchmarks and pointer operations are used in the offload regions; thus a `dim` clause cannot be used here. The `dim` clause is used in 355 and 356, which are Fortran applications and include many allocatable arrays. By comparing with Figure 7, the speedup is boosted up to 1.46x and performance did not slow down anymore after introducing `small` and `dim` clauses (note how 355.seismic overused the register files in Figure 7 and the application did slow down). Meanwhile many registers and operations are dedicated for array offset computations. The two clauses helped in reducing the number of variables used in these computations and thus improving the performance. In 356.sp, the first five kernels execution times constitute most of the benchmark execution time. Also, there are many uncoalesced memory accesses in these kernels. So the performance bottleneck is in exploiting first the memory access latency. This will require to change the benchmark algorithm which is out of scope of this work. This justifies why saving registers in these kernels did not help with improving the performance.

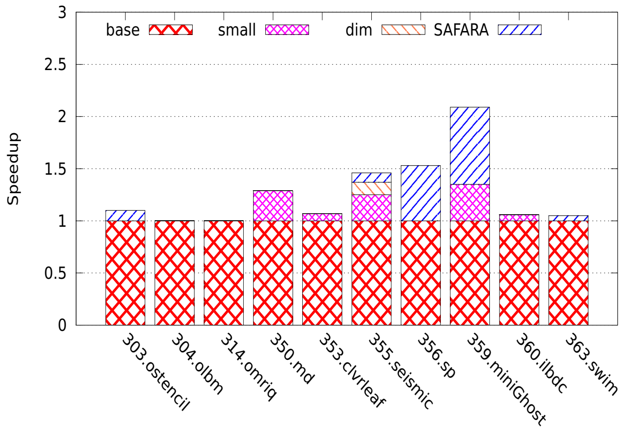

Fig. 9: SPEC ACCEL SUITE performance improvement

Figure 10 shows the performance evaluation of the NAS benchmarks. The six benchmarks are written in C language and do not use VLAs; so a `dim` clause is not useful in this case. BT, LU and SP have several kernels that contain uncoalesced memory accesses. Thus, SAFARA can help in reducing such costly accesses by prioritizing their placement in register files. However, regarding the `small` clause, among LU, SP, and BT, only BT showed benefit from using this clause. The reason is not known to us because the actual register allocation is done at a much lower level of the CUDA compiler, which we do not control.

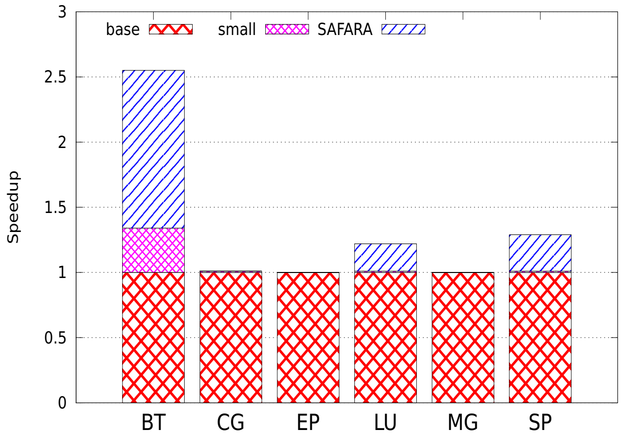

Fig. 10: NAS ACCEL SUITE performance improvement

We also compare our implementation in OpenUH with the PGI compiler for both SPEC in Figure 11 and NAS in Figure 12. We provide numbers for (1) the base OpenUH compiler by disabling the optimizations we presented in this paper, (2) enabling SAFARA, and (3) adding the two clauses to the code and then applying SAFARA. In the second and third cases, the OpenUH compiler generates efficient GPU kernels that outperform the PGI compiler. Note that the execution time in these plots is normalized in order to put all the benchmarks in the same figure; this is due to the fact that some benchmarks run in few seconds, and some of them need several hundreds of seconds to finish the execution.

$$Norm(Compiler) = \frac{ExeTime(Compiler)}{max(ExeTime(OpenUH), ExeTime(PGI))}$$

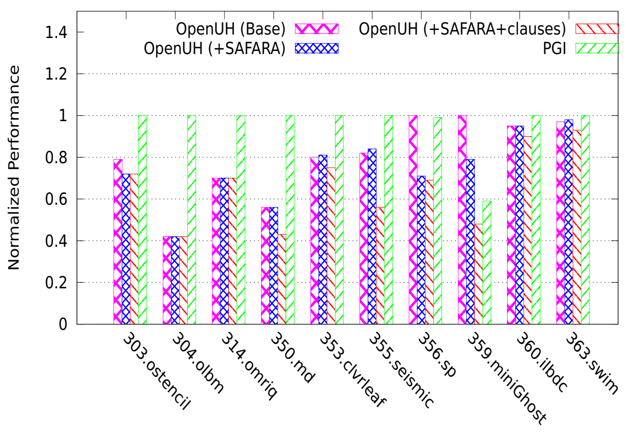

Fig. 11: SPEC performance comparison between the OpenUH and PGI compilers. The execution time is normalized and the lower, the better

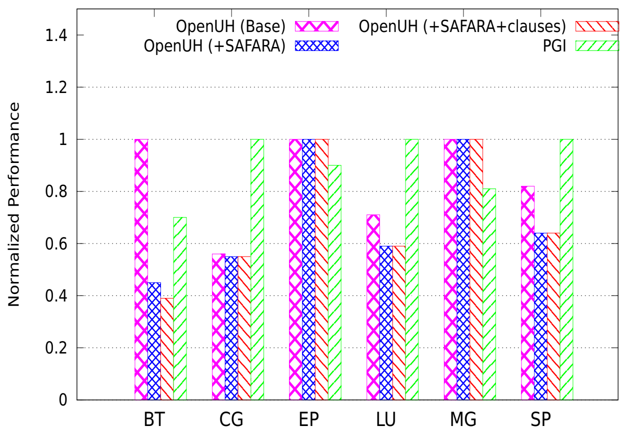

Fig. 12: NAS performance comparison between the OpenUH and PGI compilers. The execution time is normalized and the lower, the better

D. Register Usage Evaluation using `small` and `dim` Clauses

In order to assess whether the `small` and `dim` clauses can effectively reduce the register usage when multiple allocatable arrays or VLAs are used in Fortran or C codes, we introduce another metric: the number of registers used in a kernel with and without the `small` and `dim` clauses. There are 15 kernels in 355.seismic and more than 40 kernels in 356.sp. We chose the 7 hottest kernels in seismic and the 10 hottest kernels in sp to perform this experiment. Note that 355.seismic and 356.sp are two Fortran applications where allocatable arrays are used. In a compiler, there are multiple optimizations eager to use register files, such as kernel merging, loop unrolling, scalar replacement and memory vectorization [8]. The registers saved by the `small` and `dim` clauses can be used by these optimizations.

For 355.seismic, we take the 7 hottest kernels that constitute together 80% of the total execution time. Table I shows the register usage optimization results. The register files can be largely reduced by `small` and `dim` clauses if multiple allocatable arrays are used in the same kernel.

TABLE I: 355.seismic register files usage improvement via `small` and `dim` clause

Kernels	Base	+`small`	w `dim`	Saved
HOT1	128	104	48	80
HOT2	134	105	41	93
HOT3	101	90	47	54
HOT4	90	78	44	46
HOT5	86	79	44	42
HOT6	88	77	40	48
HOT7	76	73	40	36

356.sp has 10 frequently used allocatable arrays with two different dimensional information. However, most of the kernels only use one of them (the ones with NA: `dim` was not used). We chose the 10 hottest kernels (based on the execution time) and investigated the register usage information. From Table II, the register files are largely reduced in the three kernels that access multiple of these arrays. NA corresponds to kernels that use only zero, one allocatable array, or allocatable arrays that do not have equal dimensions; in this case, `dim` should not be used.

TABLE II: 356.sp register files improvement by `small` and `dim` clause

Kernels	Base	+`small`	w `dim`	Saved
HOT1	72	67	NA	5
HOT2	70	54	51	19
HOT3	82	66	NA	16
HOT4	82	66	59	23
HOT5	74	37	32	42
HOT6	57	57	NA	0
HOT7	95	78	60	35
HOT8	211	152	112	99
HOT9	184	146	114	70
HOT10	60	58	NA	2

VI. Related Work

While the original scalar replacement algorithm was proposed more than 20 years ago, computer architectures have evolved considerably since then. Numerous past works exist for improving this algorithm in many aspects. Sastry [13] and Sarkar [15] both proposed new algorithms based on the SSA form. Budiu [4] presented a simplified Carr-Kennedy [5] inter-iteration register promotion algorithm to handle a number of dynamically executed memory acceses. Hall [14] demonstrated an algorithm to increase the data reuse across multiple loops. Baradaran [3] described a register allocation algorithm that assigns registers to array references replaced with scalars along the critical paths of a computation. However, none of these algorithms can effectively work for GPU architectures. While the register moderation model in the Carr-Kennedy algorithm works well for a traditional CPU memory hierarchy, the cost model-based strategy in SAFARA selects the most profitable array reference candidates for mapping to register files through scalar replacement.

Budiu [4] proposed a simplified Carr-Kennedy iter-iteration register promotion algorithm to handle dynamically executed memory accesses. In their approach, the compiler generates a flag represented by a single bit that is associated with each value to be scalarized, as well as code that dynamically updates the flag. The flag can be inspected at run time to avoid redundant load operations, and their algorithm ensures that only the first load and last store take place. Since this algorithm inserts a large number of additional control flow statements throughout the code, the resulting behavior when executed on a GPU is thread divergence. This will produce additional overhead and significantly degrade performance. In short, this algorithm is not GPU-friendly.

Andión [2] presented a new scalar replacement algorithm for offload computation regions specified using the HMPP directive interface [7]. This work is the most similar to our paper. Both works target a GPU offload region expressed using high-level directives. Nevertheless, their algorithm over-utilizes the register files for each thread, which may cause severe performance penalties due to register spilling and GPU low threads occupancy. There are three additional limitations. First, array references with index expressions only consisting of the parallelized loop indices are potential reuse candidates. However, their approach does not handle loop-invariant variables used in array subscripts, which can also be used to estimate the reuse of array references. Second, the array reference access mode is not considered, and the cost of different types of memory access varies. For example, if a read-only array is present in the Read-Only Data Cache, then it will be accessed in a coalesced manner and it is not beneficial to replace them with scalars. Third, all the reused references are replaced with scalars. This does not take into account how many hits each reference induces. Therefore a replacement may not be beneficial in some instances when a low amount of hits occurs.

Regarding improving register usage using extensions to the API, to the best of our knowledge, our work is the first paper to propose such an optimization.

VII. Conclusion

In this paper, we present an extension to the classical scalar replacement algorithm called SAFARA that is based on feedback information regarding register utilization and a memory latency-based cost model to select which array references should be replaced by scalar references. It includes three main steps. First, a dependence analysis is used to retrieve data reuse information that is classified based on their location, and thus their latency, in the GPU memory hierarchy. Second, we use GPU tools to pinpoint the register usage information and then feed it back to the OpenACC compiler to perform the scalar replacement transformation. Third, our algorithm prioritizes the most beneficial data for allocation in registers, based on frequency of use and also memory access latency. Moreover, since the aggressive application of scalar replacement increases register pressure, we propose two new clauses to add to OpenACC, namely `dim` and `small`, to reduce the register usage.

We evaluate SAFARA and the new clauses using SPEC and NAS OpenACC benchmarks; the results suggest that these approaches are effective for improving the overall performance of code executing on the GPU. We got up to 2.5 speedup running NAS and 2.08 speedup while running SPEC benchmarks.

In future work, we plan to combine other classical optimizations like loop unrolling and memory vectorization with SAFARA and the new clauses.

ACKNOWLEDGMENT

This work was supported in part by the Total E&P USA and NSF under Award No. CCF-1409946. The authors would also like to thank Pierre Jouvelot and Gregory Rodgers whose comments improved this work and PGI for providing the compilers and support for the evaluation.

REFERENCES

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*, volume 289. Morgan Kaufmann San Francisco, 2002.

[2] J. M. Andión, M. Arenaz, F. Bodin, G. Rodríguez, and J. Touriño. Locality-Aware Automatic Parallelization for GPGPU with OpenHMPP Directives. *International Journal of Parallel Programming*, pages 1–24, 2014.

[3] N. Baradaran and P. C. Diniz. A Register Allocation Algorithm in the Presence of Scalar Replacement for Fine-grain Configurable Architectures. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 6–11. IEEE, 2005.

[4] M. Budiu and S. C. Goldstein. Inter-iteration Scalar Replacement in the Presence of Conditional Control-flow. Technical report, DTIC Document, 2004.

[5] S. Carr, S. Carr, S. Carr, K. Kennedy, K. Kennedy, and K. Kennedy. Scalar Replacement in the Presence of Conditional Control Flow. *Software Practice and Experience*, 24:51–77, 1992.

[6] G. Chen, B. Wu, D. Li, and X. Shen. Porple: An extensible optimizer for portable data placement on gpu. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 88–100. IEEE Computer Society, 2014.

[7] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[8] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):105–118, Jan 2011.

[9] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, S. Che, M. Colgrove, H. Feng, A. Grund, R. Henschel, W.-M. W. Hwu, et al. Spec accel: A standard application suite for measuring hardware accelerator performance. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 46–67. Springer, 2014.

[10] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[11] OpenACC. http://www.openacc-standard.org, 2015.

[12] OpenMP. www.openmp.org, 2015.

[13] A. Sastry and R. D. Ju. A New Algorithm for Scalar Register Promotion based on SSA Form. 33(5):15–25, 1998.

[14] B. So and M. Hall. Increasing the Applicability of Scalar Replacement. In *Compiler Construction*, pages 185–201. Springer, 2004.

[15] R. Surendran, R. Barik, J. Zhao, and V. Sarkar. Inter-iteration Scalar Replacement Using Array SSA Form. In *Compiler Construction*, pages 40–60. Springer, 2014.

[16] X. Tian, R. Xu, Y. Yan, S. Chandrasekaran, D. Eachempati, and B. Chapman. Compiler transformation of nested loops for general purpose gpus. *Concurrency and Computation: Practice and Experience*, 28(2):537–556, 2016. cpe.3648.

[17] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman. *Languages and Compilers for Parallel Computing: 26th International Workshop, LCPC 2013, San Jose, CA, USA, September 25–27, 2013. Revised Selected Papers*, chapter Compiling a High-Level Directive-Based Programming Model for GPGPUs, pages 105–120. Springer International Publishing, Cham, 2014.

[18] V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC'10*, 2010.

[19] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246. IEEE, 2010.

[20] R. Xu, X. Tian, S. Chandrasekaran, Y. Yan, and B. Chapman. Nas parallel benchmarks for gpgpus using a directive-based programming model. In *Languages and Compilers for Parallel Computing*, pages 67–81. Springer, 2014.