Interactive Refactoring for GPU Parallelization of Affine Loops

Kostadin Damevski, Madhan Muralimanohar
Virginia State University
Petersburg, VA 23806
{*kdamevski,mmadhan*}@*vsu.edu*

***Abstract*—Considerable recent attention has been given to the problem of porting existing code to heterogeneous computing architectures, such as GPUs. In this paper, we describe a novel, interactive refactoring tool that allows for quick and easy transformation of affine loops to execute on GPUs. Compared to previous approaches, our refactoring approach interactively combines the user's knowledge with that of an automated parallelizer to produce parallel CUDA GPU code. The generated code retains the structure of the original loop in order to remain maintainable. The refactoring tool also computes and displays profitability metrics, intended to advise the user of the performance potential of the generated code.**

***Keywords*-**

I. Introduction

Refactoring is a software engineering technique, whereby a program's structure is modified without any change to its function. Common refactoring techniques, such as extracting or inlining a method, or renaming a field, have been in practical use for a number of years with the aim of improving software quality [1]. Integrated Development Environments (IDEs) (e.g. Eclipse, NetBeans) host rich sets of refactoring utilities, however their use within the HPC community is not widespread. Several new projects, such as the Eclipse Parallel Tools Platform (PTP) [2] have recently made strides to adapt IDEs to the needs of High-Performance Computing (HPC) developers.

In IDEs, refactoring is performed interactively, where the programmer selects the refactoring target, changes the configuration parameters of the transformation, and is shown an overview of the changes that will take place to the original program. The programmer is also allowed to undo the refactoring and revert the program state at the click of a button. Interactivity and tight integration with an IDE can significantly improve the refactoring success, leading to higher usability of a refactoring tool [3].

This paper applies interactive refactoring to the problem of parallelizing loops with affine iteration and array access patterns to use NVIDIA's Compute Unified Device Architecture (CUDA) programming model. Our novel refactoring, called EXTRACT KERNEL, is implemented as a plugin to the broadly used Eclipse development environment. The tool performs automatic parallelization and alleviates the programmer from writing dozens of lines of low-level, tedious, and error prone code.

In the scientific and high-performance computing domains, porting to new platforms has been shown to be one of the largest barriers to high programmer productivity. Several studies cite the difficulty and time overhead in porting scientific code to the latest class of supercomputers [4], [5], [6]; time that could have been spent in developing new functionality and speeding the path to new scientific discoveries in a number of disciplines. Our tool targets programmers like these, who are familiar enough with CUDA to understand and maintain the refactored code, but could use the help of a tool to quickly transform a larger body of code. To support these programmers, EXTRACT KERNEL's priority is to generate code similar to human-written CUDA code in which the programmer can easily recognize the original loop body.

Automatically parallelizing compilers have long ago established the theory and practice for parallelizing affine loops, a process performed without the involvement of the programmer. However, the programmer's domain knowledge is often required in order to provide a more effective parallelization, based on the particular problem and usage scenario. Interactive parallelization tools, such as ParaScope [7] and SUIF Explorer [8], take a complimentary approach in integrating the parallelizing compiler with the user's knowledge of the problem. We extend the work of these interactive approaches in providing a program refactoring that aids the programmer to arrive to a parallel program that executes on a GPU.

This paper makes the following contributions:

- Description of the problem of transforming sequential C loops into CUDA parallel code that does not obfuscate the original loop body.
- An transformation algorithm implemented in the EXTRACT KERNEL tool and distributed as an Eclipse plugin.
- A technique, based on arithmetic intensity, to advise programmers about loops which will perform poorly on a GPU.
- Evaluation of the transformation and performance advisor, based on the design goals, is performed.

II. Motivating Example

In this section we give an overview of refactoring loops into CUDA kernels by using a simple loop that performs a

```
1. #define N 1000
2.
3. int main()
4. {
5.      float x[N], y[N], a;
6.
7.      for(int i=0; i < N; i++) {
8.           y[i] = a*x[i] + y[i];
9.      }
10.}
```

(a)

```
1. __global__ void gpu_axpy(float a, float *x, float *y)
2. {
3.      int idx = blockIdx.x * blockDim.x + threadIdx.x;
4.      if(idx < N)
5.           y[idx] = a * x[idx] + y[idx];
6. }
7.
8. int main()
9. {
10.     float x[N], y[N], a;
11.
12.     float *x_d;
13.     cudaMalloc((void **) &x_d, sizeof(float)*N);
14.     cudaMemcpy(x_d, x, sizeof(float)*N, cudaMemcpyHostToDevice);
15.     float *y_d;
16.     cudaMalloc((void **) &y_d, sizeof(float)*N);
17.     cudaMemcpy(y_d, y, sizeof(float)*N, cudaMemcpyHostToDevice);
18.
19.     int numThreadsPerBlock = 256;
20.     int numBlocks = N / numThreadsPerBlock + 1;
21.     gpu_axpy<<<numBlocks,numThreadsPerBlock>>>(a, x_d, y_d);
22.
23.     cudaMemcpy(x, x_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
24.     cudaFree(x_d);
25.     cudaMemcpy(y, y_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
26.     cudaFree(y_d);
27.}
```

(b)

Figure 1. AXPY kernel in C (part a) and CUDA (part b).

AXPY (Alpha X Plus Y), $\alpha x + y$, operation on two vectors of equivalent size N, x and y (see Figure 1(a)).

CUDA introduces a minor extension of the C language and a set of libraries exposed through conventional API calls. To a CUDA programmer, a program consists of two parts: one that executes on the CPU (or host) and one that executes on the GPU (or device). The GPU part of the code consists mainly of data-parallel functions, called *kernels*. To use the GPU, CUDA code follows the following workflow:

1) Allocate and copy necessary data into the GPU memory (Figure 1(b), lines 12-17).
2) Specify the number of threads and launch the kernel (Figure 1(b), lines 19-21).
3) Execute the kernel function, placing the results in globally accessible memory (Figure 1(b), lines 1-6).
4) Copy results back to the CPU memory and free memory on the GPU (Figure 1(b), lines 23-26).

The EXTRACT KERNEL refactoring analyzes sequential loop code and transforms it into equivalent parallel CUDA kernel (i.e. Figure 1(a) to Figure 1(b)). The workflow of this refactoring is depicted in Figure 2. The refactoring begins when the user concurrently selects a loop (e.g. lines 7-9 in Figure 1(a)) and chooses the "Extract to CUDA Kernel" menu option in the Eclipse environment. The selected loop is extensively analysed to determine whether it passes a set of preconditions specific to this refactoring. If the preconditions are met, an initial screen is presented to the user that enables the selection of the kernel name, and the tuning of GPU platform parameters (e.g. the maximum number of threads per block). This is followed by a refactoring preview screen that clearly and in graphical form outlines the modifications to the original code. Once the user gives his or her final approval, the refactoring takes place. All refactorings in Eclipse are easily reversible if the user is not satisfied with the end result. Also, if a candidate loop fails to pass one of the preconditions, the user is presented with an informative error message detailing the reason for rejecting the loop (e.g. the loop contains a data dependence on a specific variable). Once the user fixes this problem, the refactoring can be reinitiated.

III. EXTRACT KERNEL REFACTORING

In this section we present the design of the EXTRACT KERNEL refactoring, which transforms sequential C loops into parallel CUDA code. EXTRACT KERNEL strives to achieve high usability by providing an interactive approach to refactoring, in contrast to most other program transformation tools for parallel and distributed computing, which often rely on scripts or annotations to the source code [9], [10]. In addition, we anticipate that the user of this refactoring will maintain and modify the generated CUDA code. Therefore, EXTRACT KERNEL's philosophy is to generate code that

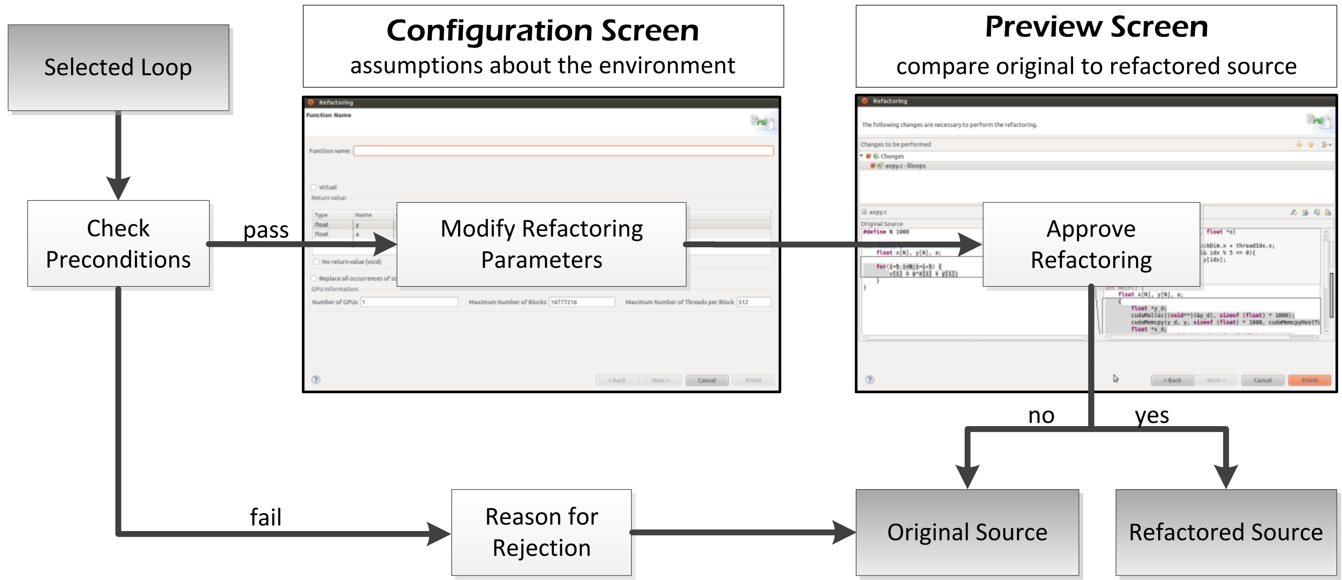

Figure 2. The refactoring workflow follows a set of tasks through configuration and preview screens, which are common to most Eclipse refactorings.

closely follows the code style of a human CUDA developer. To accomplish this. we require that the generated code, within the CUDA kernel, follows the contents of the original loop, and avoid any transformations that obfuscate the original loop's contents. In following this design principle, we consciously refuse to perform loop transformation that may improve performance [1] or remove data dependencies. To the same end, EXTRACT KERNEL parallelizes the outermost loop of a nest, keeping the inner loops the same when executing on the GPU [2]. While the outermost loop is the only one refactored, the dependence testing still takes notice of nested loops to assure the transformation is safe.

Below we discuss some of the challenging static analysis tasks required by EXTRACT KERNEL, followed by the set of preconditions a candidate loop must satisfy before refactoring. A number of design choices, with various trade-offs, exist in generating the CUDA code. These choices, coupled with the decisions we made in EXTRACT KERNEL are discussed in the last part of this section.

A. *Analysis*

Determining whether a loop is parallelizable. First, the proposed refactoring must determine whether the candidate loop contains data dependencies constraining its parallelization. Loop-carried dependencies, where a data dependence exists between separate iterations of the loop, require analysis of the interplay between loop induction variables and array subscripts. Algorithms for detecting loop-carried dependencies (e.g. GCD test, Banerjee test) have been established by previous research in automatically parallelizing compilers [11], and can generally be applied to this refactoring.

Understanding the data structure and size. In order to generate code that copies the data structures between CPU and GPU memories, the structure of the data must be extracted, a process that can be very challenging for deeply nested or aliased structures. A related challenge in the proposed refactoring is determining the size of the data that should be transferred, which is required by the calls to `cudaMalloc` and `cudaMemcpy`. If the data size is not available as a literal, the refactoring can generate code that uses a symbolic representation for the size. In certain loops, however, complicated interprocedural dataflow analysis may be required to determine the size of a particular variable, and the general case reduces to Turing's halting problem and is undecidable.

Mapping loop iteration space to GPU thread space. In parallelizing a loop, the number of loop iterations, extracted in the form of a literal, symbol or simple expression, is mapped to the number of parallel threads. Typical CUDA programs create very large numbers of threads, and therefore, a one-to-one mapping between loop iterations and threads is reasonable for EXTRACT KERNEL. However, hard limits for the number of threads imposed by the GPU hardware should not be exceeded by the refactored code. In CUDA, these architectural limits are available via an invocation to the `CudaGetDeviceProperties` method. In similar fashion, limits for the maximum allowable threads

[1]Loop transformations (e.g. loop tiling, unrolling, etc.) are not guaranteed to improve performance, and may in fact have the opposite effect. Determining the optimal sequence of transformations to apply is an active area of research.

[2]Alternatively, it is possible to replace the entire loop nest with a CUDA thread block, and add a `cudaThreadSynchronize` invocation between the inner and outer loop threads.

per thread block may be used to allocate threads into blocks, as in lines 19-20 of Figure 1(b).

B. Refactoring Preconditions

EXTRACT KERNEL evaluates candidate loops to determine whether they satisfy its preconditions. Only the loops that satisfy these preconditions can be safely refactored, resulting in code that is guaranteed to perform the same task as the original loop, in parallel and on the GPU. The necessary preconditions are the following:

- The loop has affine iterations and affine array access patterns.
- The number of loop iterations can be statically determined.
- The referenced data elements do not overlap in memory or alias each other.
- No `break` or `return` statements in the candidate loop body.
- No method calls in the candidate loop body.
- The loop does not contain data dependencies inhibiting its parallelization.

Affine loops are based on loop iterations and array accesses that are affine functions of surrounding loop variables, constants, and program parameters. Such loops are commonly encountered in practice. Affine loops are frequently normalized (cannonicalized) into a standard form that is easier to parse and analyse by compilers and program transformation tools.

Certain loops may contain statements that make the estimation of the number of loop iterations difficult or impossible to perform statically. For instance, a loop body that dynamically changes the loop induction variable using a process that depends on variables that change at runtime. This type of loops are not parallelizable, as it is very difficult or impossible to correctly determine the data dependence, or to map the iteration space of the loop to GPU threads.

Accesses to deeply nested or aliased data structure are difficult to follow, and difficult to test for data dependence. Static analysis of such structures in permissive languages such as C is an area of research and development. A refactoring tool could, in the future, rely on a state-of-the-art static analysis toolkit that would allow greater flexibility in analysing data structures.

The `return` statement cannot be safely moved into a new function, such as a GPU kernel. Refactoring this statement into a new method would result in code that does not follow the original execution path. This is also a correctness check performed by the standard EXTRACT METHOD [1] (EXTRACT FUNCTION) refactoring. The `break` statement causes a similar problem to parallelization as dynamically changing the loop iteration pattern: it is impossible to undo the work that parallel threads have already completed upon encountering a `break` statement.

CUDA kernels are not permitted to invoke any CPU functions, so EXTRACT KERNEL simply disallows the existence of method calls in the candidate loop body. This has the added effect of making the analysis that the EXTRACT KERNEL tool has to perform a lot simpler - by not having to follow the call hierarchy. Within Eclipse, the user can attempt to use the INLINE METHOD [1] refactoring before retrying EXTRACT KERNEL.

The analysis of whether the loop contains data dependencies is necessary for its parallelization. Intuitively, parallelism reorders the operations executed in the original loop, which is permissible only if such reordering does not affect the loop's output. Data dependence testing is undecidable for any general loop. However, for the class of affine loops, data dependence algorithms have been established based on finding solutions to linear diophantine equations. In EXTRACT KERNEL we initially rely on the GCD (Greatest Common Divisor) test to conclusively determine that no dependence exists. If this test cannot prove independence, we use approaches from integer linear programming that culminate with a branch-and-bound algorithm for solving general integer linear programming problems. This dependency testing approach follows general approaches common in parallelizing compilers [12]. Minor differences in the algorithm implementations are due to EXTRACT KERNEL parallelizing only the outermost loop, instead of the entire loop nest.

IV. CUDA CODE GENERATION

EXTRACT KERNEL gives priority to producing code which does not modify the structure of the original loop. In other words, the refactoring attempts to simply move the contents of the loop into the CUDA kernel, while making few changes, and allowing the programmer to easily recognize his or her original intent. However, this is not easily accomplished in a complex refactoring, such as EXTRACT KERNEL, in which the loop body needs to be parallelized and written in CUDA. Several code generation trade-offs exist, where the simplest code may not be the most efficient in terms of resource utilization or performance. Apart from the parallel kernel, the refactoring generates blocks of code to transfer data to and from the GPU, code to calculate the number of threads and blocks, and to launch the kernel.

A. Kernel Generation

Refactoring a loop into a parallel kernel function requires that the loop statement is removed, while the body of the loop becomes the body of the function. Within this new kernel function, the loop index variable is replaced by the `threadID`, as each loop iteration is allocated to an individual thread. This design decision is consistent with getting good performance on a GPU, as array accesses that are a simple affine function of the loop index variable can be coalesced. Coalesced memory accesses, where adjacent threads within a GPU half-warp access memory that is aligned to 4,8 or 16 bytes, can be optimized by the hardware to experience much lower memory access latency [13].

Another modification to the original loop's body is the replacement of `continue` statements in the loop with `return` statements in the resulting parallel function. This is consistent with the expected semantics of loops, whereby an iteration is terminated by the `continue` statement, which corresponds to a thread returning from the execution of a parallel kernel.

Loops that contain an increment greater than 1 can be mapped into thread space in two different ways: 1) by tightly mapping each loop iteration into a thread and adding code that normalizes references to loop index variable (i.e. `threadID`); or 2) by loosely mapping and launching some threads that will perform no actual work. Option 1 produces better resource usage at the cost of modifying the code in the loop (kernel) body, which is against EXTRACT KERNEL's design principle. Option 2 keeps the loop body code clean, but can be extremely inefficient with respect to thread usage. EXTRACT KERNEL can generate code according to either of these two options, allowing the programmer to be the final arbiter. The programmer is presented with this choice via a check box in the configuration screen.

To illustrate the difference between the two kernel generation options, consider the simple loop below, whose iteration space is defined by the $first$, $last$ and $stride$ variables. The loop body references a one-dimensional array A.

```
for(int i=first; i<last; i+=stride)
{
  A[i]++;
}
```

The first kernel generation option, where the generated kernel has an efficient utilization of its threads, takes the following form:

```
__global__ void tight_pack(float *A)
{
  int idx = blockIdx.x * blockDim.x
            + threadIdx.x;
  idx = idx + first;
  A[idx * stride]++;
}
```

This kernel is invoked with number of threads equivalent to the number of loop iterations: $(last - first)/stride$. On the other hand, the generated code for the loosely-packed (second) option is invoked with $last - first$ threads. The generated code of the loosely-packed kernel follows:

```
__global__ void loose_pack(float *A)
{
  int idx = blockIdx.x * blockDim.x
            + threadIdx.x;
  idx = idx + first;
  if(idx % stride == 0) {
    A[idx]++;
  }
}
```

The second option does not complicate the array subscript expression, but is likely to result with many threads failing to enter the body of the conditional statement, and performing no work. This does not have significant performance implications, but it can result in certain loops too rapidly reaching the architectural ceiling on the number of threads.

B. Data Transfer Code Generation

Data transfer to the GPU and back requires invocations to the `cudaMalloc`, `cudaFree` and `cudaMemcpy` functions. These functions follow the structure of their relatives in the C standard library, with the difference that they produce pointers to GPU memory. Dereferencing these pointers from the CPU would have adverse effects. In transferring multidimensional arrays to GPU memory, we create an array of pointers to GPU memory. Pointers to this array cannot be dereferenced in order to allocate space for the subarrays that will contain the actual data. Rather, data has to carefully be copied into this array. This is a careful process known to skilled CUDA programmers, but one that generally tricks novices and where EXTRACT KERNEL's code generation can be helpful to programmers.

1) Inferring Unavailable Array Size: One of the key pieces on information necessary in generating correct invocations to `cudaMalloc` and `cudaMemcpy` is the size of an array. However, in practice, we rarely encounter statically-allocated arrays that are located in a recent stack frame, whose size can easily be inferred via static analysis of the program. Instead, we often see arrays passed into the current function as pointers, where simple program analysis cannot easily find the line on which the array was defined and allocated.

In many of these cases, analysis of the loop iteration and array subscript expression(s) can shed enough light on the size of the array for EXTRACT KERNEL's purposes. For simple array subscript expressions, which involve only constant factors of the loop induction variable, the bounds of the array accesses can be inferred via the loop bounds. The loop bounds, in turn, are determined via projections to one axis in the polyhedral loop model, and may be computed via Fourier-Motzkin elimination. Based on such analysis of the loop, EXTRACT KERNEL can determine that an array is at least as big as the part that is referenced inside the loop. Using this assumption, EXTRACT KERNEL can proceed with the generation of appropriate memory transfer function calls.

This approach can even be seen as an optimization over blindly transferring the entire array, in cases where only part of the array is accessed inside the loop.

C. Kernel Launch Code Generation

The number of threads launched by EXTRACT KERNELS equals the number of iterations of the refactored loop. Hard limits on the number of threads for a particular GPU architecture have to be obided by, or the kernel cannot be launched. Instead of inserting runtime invocations to `cudaGetDeviceProperties`, the programmer is allowed to enter the number of threads in the configuration screen of the refactoring, while a reasonable default value is already present. Entering this hardware configuration constant once for the entire use of a particular GPU is a reasonable requirement to ask of a programmer.

The programmer is also asked to enter a constant for the maximum number of threads per block. EXTRACT KERNEL's block mapping is simple, and fills up each block to the maximum number of threads before proceeding to the next one. As the tool only refactors a single loop, the outermost in a loop nest, it cannot easily complicate the way work in a loop nest is allocated to blocks of threads.

V. PERFORMANCE ADVISOR

Many examples exist where parallel GPU code does not execute faster than sequential CPU code, often due to the significant GPU memory transfer cost. To help the programmer decide whether refactoring a particular loop would be profitable, EXTRACT KERNEL provides a refactoring advisor based on two related, statically-computed metrics: arithmetic intensity and amount of work. These metrics, coupled with information regarding their interpretation, are displayed within EXTRACT KERNEL's refactoring workflow.

The amount of work of a loop/kernel is its number of arithmetic operations. Arithmetic intensity is the number of arithmetic operations (i.e. amount of work) versus the cost of transferring the necessary data to and from the device [14]. Intuitively, the higher the arithmetic intensity and amount of work, the more likely that a particular loop will perform well on the GPU. To validate the relationship of these metrics to the speedup obtained by executing on the GPU, we performed an experiment, whose results are shown in Figure 3. The experiment is based on a synthetic loop/kernel that was executed on an Intel Xeon quad-core CPU and a NVIDIA Quadro 2000 GPU.

The experimental results show that relatively high levels in both of the two metrics are necessary to achieve speedup. The reason for this is that, first, the arithmetic intensity needs to be significant enough to overcome the memory transfer cost. Second, the amount of work needs to be large enough to hide the cost of starting the kernel, and to accumulate a performance benefit compared to serial CPU execution. This

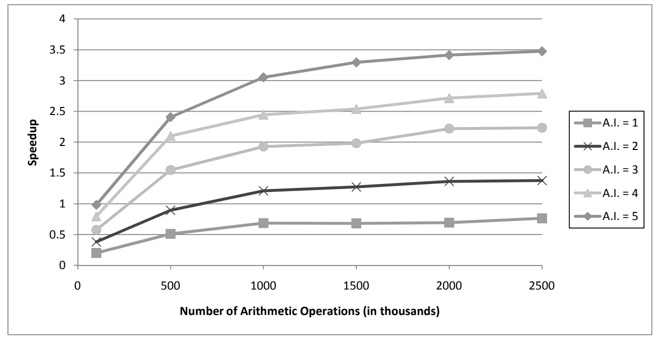

Figure 3. Experiment of arithmetic intensity using a synthetic kernel/loop. A combination of significant arithmetic intensity and number of operations is needed to achieve a reasonable speedup in the GPU vs. the CPU.

conclusion is consistent with early work on the performance of GPU for general purpose processing [15].

EXTRACT KERNEL uses the following equations to calculate arithmetic intensity and amount of work, using inputs acquired via static analysis of the candidate loop:

$$amt.work = \sum_{i=1}^{n} ops_i * \prod_{j=1}^{i} iters_j$$

$$arith.intensity = \frac{amt.work}{datasize * 2}$$

In the above, n is the number of nested loops, $iters_i$ is the number of iterations in loop i, assuming that loops in a loop nest are ordered from outermost to innermost. The number of arithmetic operations in loop i, that do not also belong to any loop that is inner to i, are denoted as ops_i. The $datasize$ is the number of elements of all the variables that are referenced in the loop nest; a value that is already necessary and computed by the core aspects of the EXTRACT KERNEL refactoring. The ops_i can easily be extracted for simple loops. On the other hand, loops that contain more than one control-flow path require that each path is considered in isolation. In such cases, instead of presenting the performance metrics for each control-flow path, we opt to show only the maximum and minimum values, assuming that the range is sufficient in providing a performance estimate of the refactored code. For certain complex loops that integrate several conditional and iteration constructs, it may be difficult to ascertain the minimum and maximum ops_i, amount of work, and arithmetic intensity. For those loops EXTRACT KERNEL offers no performance advice, as the computed metrics may confuse or seriously mislead the user.

Further, in EXTRACT KERNEL, as only static program information is available, it is often impossible to calculate arithmetic intensity and amount of work exactly (i.e. as a literal value). Information about the number of loop iterations and the number of data elements is very often in symbolic (variable) form. Despite this, we believe that the arithmetic intensity expressed a combination of symbolics and literals can be of use to the programmer in deciding whether to refactor. The programmer can often understand the meaning of a simple function, involving symbolic variable names taken from a program he or she is familiar with, as long as the function does not contain a large number of unknowns.

Apart from the above, an additional set of simplifying assumptions exist in the design of the performance advisor, such as, for example, treating floating point and integer operations as equivalent, or ignoring the mapping of thread block to processing units. Despite the seemingly large number of assumptions, we find the performance advisor to be a useful part of EXTRACT KERNEL, especially in cases where it presents overwhelming evidence not to, or to refactor.

VI. RESULTS

In evaluating this refactoring we would like to answer the following set of questions:

1) Is EXTRACT KERNEL applicable?
2) Does the performance advisor produce accurate and interpretable refactoring profitability estimates?
3) Is EXTRACT KERNEL useful in improving the performance of the refactored application?

To determine whether EXTRACT KERNEL can be applied to practical problems, we applied it to a large number of loops in well-known applications from the scientific and high performance computing domain, the GNU Scientific Library (GSL) and the LAMMPS Molecular Dynamics Simulator [3]. Table I shows the results of this experiment. While approximately one third of all of the affine loops were processable by EXTRACT KERNEL, a few avenues of improvement are apparent.

A large percentage of loops were rejected because of a function call in the loop body. One way of improving on this is for the user (or some automated facility) to apply the INLINE FUNCTION refactoring before applying EXTRACT KERNEL. Another approach can be to transform the called function to execute on the GPU as well.

A significant portion of the affine loops are not processable by EXTRACT KERNEL due to the inability to determine the size of one or more variables referenced in loop body. This metric includes the assumption/optimization of using loop iterations to determine the sizes of certain arrays. Enhancements to program analysis techniques utilized by EXTRACT KERNEL (e.g. interprocedural dataflow analysis) is likely to yield a reduction in the number of rejected loops due to unknown data size. Bridging fully featured compiler middle-ends to Eclipse, such as those in Rose [16] or LLVM [17], is likely the easiest path to availing sophisticated static program analysis algorithms.

[3]LAMMPS already contains some CUDA code. Our experiments were performed on other portions of its substantial code base.

	GSL	LAMMPS
Total Source LOC	170338	207195
# Compilation Units	999	668
# For Loops	3303	5760
Loops rejected b/c function call	2084 (63%)	1291 (22%)
Affine Loops	606 (18%)	3078 (53%)
Loops with data dependence	319 (53%)	1517 (49%)
Loops with unknown data size	99 (16%)	426 (14%)
Loops satisfying all preconditions	188 (31%)	1135 (37%)
Avg. variables in arith.intensity	1.44	0.53

Table I
STUDY OF LOOPS FOUND IN TWO SCIENTIFIC APPLICATIONS. WE COUNTED THE NUMBER OF FOR LOOPS, THOSE THAT WE WERE ABLE TO SUCCESSFULLY PARSE BECAUSE THEY WERE AFFINE, AND THOSE THAT WERE ALSO PARALLELIZABLE AND FULLY PROCESSABLE BY EXTRACT KERNEL. WE ALSO SHOW THE NUMBER OF LOOPS REJECTED DUE TO A FUNCTION CALL IN THE LOOP BODY AND DUE TO UNKNOWN DATASIZE.

GSL Loop	Arithmetic Intensity
histogram	$\frac{3*(n+1)}{(n+7)*2} \approx 1.5$
randdist	$\frac{kEvents}{(2*kEvents+3)*2} \approx 0.25$
ode	$\frac{5*dim}{(4*dim+3)*2} \approx 0.625$
specfunc	$\frac{2*nmax}{(nmax+6)*2} \approx 1$

Table II
THE ARITHMETIC INTENSITY OF THE LOOPS BENCHMARKED IN FIGURE 4

To assess the interpretability of EXTRACT KERNEL's performance advice, we determined the average number of variables present in the arithmetic intensity metric when computed on the refactorable loops in the GSL and LAMMPS code bases. A large number of variables hurts interpretability by increasing the difficulty in deciding whether or not to refactor. The results of this experiment are shown in the bottom of Table I, and the indication is that the average number of variables was reasonably low in both GSL and LAMMPS. In addition, those cases where more than one variable was present in the arithmetic intensity metric, the variables were overwhelmingly clustered in the bottom ($datasize$) part of the arithmetic intensity equation. Compared to variables on both sides of the equation, such arithmetic intensity can be somewhat easier to interpret.

The final experiment had the aim of determining the accuracy of the performance advisor and, also, validating the proposition that refactored code can achieve a performance improvement. To show this, we sampled five loops in the GSL, ranging in the type of function they performed (computing ordinary differential equations, probability, spectral methods etc.), and refactored them using EXTRACT KERNEL. The speedup achieved, compared to the original, unrefactored code is reported in Figure 4, for different input data sizes.

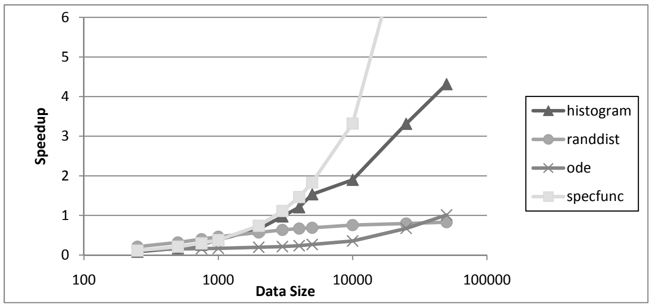

Figure 4. Speedup of refactored GPU code, compared to original CPU code, for a few selected GSL loops.

VII. Related Work

In this paper, we propose a refactoring technique to help address the porting difficulties in HPC software development. Porting code to a new platform does not change its observable runtime behavior, and therefore fits well within the definition of refactoring. Kjolstad et al. [18] foresee the widespread use of IDEs by HPC programmers, and propose of a number of new refactoring techniques targeted for this community.

Relooper is an interactive tool that parallelizes Java loops by refactoring them to use the new Java `ParallelArray` construct. A number of ReLooper's ideas are followed in the design of EXTRACT KERNEL. However, several other differences between the two tools exist. For instance, ReLooper is required to follow the method call path of the refactored code, while our tool does not as GPU code can not make method calls to other CPU methods, allowing us to simply reject those refactoring candidates. In addition, ReLooper ensures that a loop is sequential, which is not a constraint in CUDA.

A number of high-level abstractions have recently been proposed to reduce developer effort in writing CUDA programs [19], [20], [21] and improve program portability to a wider range of parallel platforms. While having the potential to improve productivity in programming hybrid architectures, these approaches do not address legacy code. To use these tools existing code must be rewritten, which requires extra effort and rarely gains significant momentum in the developer community. EXTRACT KERNEL is appropriate for development based on a legacy code base, and, unlike the other approaches, it also ensures a degree of safety in the resulting parallel program.

Aspect-Oriented Programming (AOP) was proposed by Wang and Parashar [22] as a generative method of abstracting some of the details in CUDA programs. While this approach succeeds at greatly simplifying the development of CUDA programs, it again requires that the user learns a new set of higher-level primitives before he or she can become productive. A few other tools, aimed at porting legacy code bases to use the GPU, have also been proposed [9], [10]. While these tools have similar goals as EXTRACT KERNEL, they attempt to achieve them in different ways.

Both interactive parallelization and automated porting to GPUs have been attempted before (e.g. ReLooper, CUDA-Lite). However, we are not aware of any other interactive approach to GPU parallelization, such as the one presented in this paper.

VIII. Conclusions and Future Work

This paper presents the EXTRACT KERNEL interactive refactoring and Eclipse plugin, which transforms affine loops into GPU kernels and their associated memory copy and invocation code. EXTRACT KERNEL can increase the productivity of a programmer in transforming existing serial code to use NVIDIA's CUDA parallel programing environment. The refactoring is aimed as an aid to programmers that are familiar with CUDA, as it attempts to generate readable code and offers no support for maintaining the generated code. The performance advisor, which is part of EXTRACT KERNEL, computes and displays statically computed performance metrics that indicate the profitability of the refactoring. The refactoring is evaluated and our results indicate that it achieves a set of relevant goals.

A few improvements of EXTRACT KERNEL are in our sight, as the future work of this project. We intend to produce the opposite refactoring that would inline a GPU kernel as a serial CPU loop. Also, we intend to enhance EXTRACT KERNEL to handle function calls in the loop body, by producing a separate GPU kernel for each invoked function. Finally, we would like to enhance the applicability of the tool by including more sophisticated static analysis algorithms, such as interprocedural control flow analysis.

Acknowledgment

The authors gratefully acknowledge support for this work from the U.S. Department of Energy through award number DE-SC00005371, and the Thurgood Marshall College Fund.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, 1st ed. Addison-Wesley Professional, Jul. 1999.

[2] Eclipse Parallel Tools Platform (PTP), "http://www.eclipse.org/ptp," accessed August 2011.

[3] D. B. Roberts, "Practical analysis for refactoring," Ph.D. dissertation, Champaign, IL, USA, 1999.

[4] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, "Software development environments for scientific and engineering software: A series of case studies," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 550–559.

[5] S. Faulk, E. Loh, M. L. V. D. Vanter, S. Squires, and L. G. Votta, “Scientific computing’s productivity gridlock: How software engineering can help,” *Computing in Science and Engineering*, vol. 11, pp. 30–39, 2009.

[6] S. Squires, M. Van De Vanter, and L. Votta, “Yes, there is an ’expertise gap’ in hpc application development,” in *Proceedings of the 3rd International Workshop on Productivity and Performance in High-End Computing (PPHEC ’06)*. IEEE CS Press, 2006.

[7] M. W. Hall, T. J. Harvey, K. Kennedy, N. McIntosh, K. S. McKinley, J. D. Oldham, M. H. Paleczny, and G. Roth, “Experiences using the ParaScope Editor: an interactive parallel programming tool,” in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP ’93. New York, NY, USA: ACM, 1993, pp. 33–43.

[8] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam, “SUIF Explorer: an interactive and interprocedural parallelizer,” in *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 1999.

[9] S. Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W. Mei, “Languages and Compilers for Parallel Computing,” J. N. Amaral, Ed. Berlin, Heidelberg: Springer-Verlag, 2008, ch. CUDA-Lite: Reducing GPU Programming Complexity, pp. 1–15.

[10] S. Lee, S. J. Min, and R. Eigenmann, “OpenMP to GPGPU: a compiler framework for automatic translation and optimization,” *SIGPLAN Not.*, vol. 44, pp. 101–110, Feb. 2009.

[11] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2001.

[12] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, 2nd ed. Prentice Hall, Sep. 2006.

[13] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Z. Ueng, J. A. Stratton, and W.-M. Hwu, “Program Optimization Space Pruning for a Multithreaded GPU,” in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO ’08. ACM, 2008, pp. 195–204.

[14] M. Harris, “Mapping computational concepts to GPUs,” in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH ’05. ACM, 2005, pp. 50+.

[15] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, “Brook for GPUs: stream computing on graphics hardware,” *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 777–786, Aug. 2004.

[16] C. Liao, D. Quinlan, R. Vuduc, and T. Panas, “Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization Languages and Compilers for Parallel Computing,” ser. Lecture Notes in Computer Science, G. Gao, L. Pollock, J. Cavazos, and X. Li, Eds. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2010, vol. 5898, ch. 21, pp. 308–322.

[17] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, ser. CGO ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75+.

[18] F. Kjolstad and M. Snir, “Bringing the HPC Programmer’s IDE into the 21st Century through Refactoring,” in *SPLASH 2010 Workshop on Concurrency for the Application Programmer (CAP’10)*. Association for Computing Machinery (ACM), Oct. 2010.

[19] D. M. Kunzman and L. V. Kalé, “Towards a framework for abstracting accelerators in parallel applications: experience with cell,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC ’09, 2009.

[20] OpenCL - The open standard for parallel programming of heterogeneous systems, “http://www.khronos.org/opencl,” accessed January 2011.

[21] NVIDIA Thrust GPU Library, “http://code.google.com/p/thrust,” accessed March 2011.

[22] M. Wang and M. Parashar, “Object-oriented stream programming using aspects,” in *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010.