PSkel: A Stencil Programming Framework for CPU-GPU Systems

Alyson D. Pereira  
Department of Computer Science  
PUCMinas, Brazil  
avyson.pereira@sga.pucminas.br

Luiz Ramos  
Institute of Computing  
Unicamp, Brazil  
lui.ramos@ic.unicamp.br

Luís F. W. Góes  
Graduate Program in Electrical Engineering  
PUCMinas, Brazil  
lfwgoes@pucminas.br

Abstract

The use of Graphics Processing Units (GPUs) for high-performance computing has gained growing momentum in recent years. Unfortunately, GPU-programming platforms like CUDA are complex, user unfriendly, and increase the complexity of developing high-performance parallel applications. In addition, runtime systems that execute those applications often fail to fully utilize the parallelism of modern CPU-GPU systems. Typically, parallel kernels run entirely on the most powerful device available, leaving other devices idle. These observations sparked research in two directions: (1) high-level approaches to software development for GPUs, which strike a balance between performance and ease of programming; and (2) task partitioning to fully utilize the available devices. In this paper, we propose a framework, called PSkel, that provides a single high-level abstraction for stencil programming on heterogeneous CPU-GPU systems, while allowing the programmer to partition and assign data and computation to both CPU and GPU. Our current implementation uses parallel skeletons to transparently leverage Intel TBB and NVIDIA CUDA. In our experiments, we observed parallel applications with task partitioning can improve average performance by up to 76% and 28% compared to CPU-only and GPU-only parallel applications, respectively.

1 Introduction

In recent years, Graphics Processing Units (GPUs) have been used in conjunction with general-purpose CPUs to enable high computing performance with high energy efficiency. While modern CPUs use large caches and provide multiple out-of-order cores with branch prediction and speculation, GPUs are much richer in floating point units and provide large amounts of simple processing cores [1].

Despite being commonly found in the same hardware platform or even on the same chip, CPUs and GPUs typically have different application programming interfaces. In fact, widespread programming approaches for GPUs provide very low-level programming abstractions. Furthermore, most of them require knowledge and careful use of the GPU memory model to maximize parallelism. That complexity compounds the inherent complexities of programming parallel applications, which requires reasoning about computation and coordination steps. Thus, programming parallel applications for CPU-GPU platforms can prove challenging, tedious, and error-prone, even for experienced programmers [2].

A common approach to address the CPU-GPU programming complexity is the use of algorithmic skeletons. Parallel skeletons model and abstract common parallel programming patterns (computation and coordination phases), thereby enabling the programmer to focus on algorithm design, rather than on runtime system details. Among existing parallel skeletons, the stencil pattern is critical in many scientific computing domains, including image and signal processing and computational fluid dynamics [3, 4]. The large body of recent work targeting GPU implementations of high-performance stencil computations stresses the importance of that pattern [5–8].

Another important aspect of CPU-GPU platforms is that their runtime systems generally fail to exploit the platform’s full potential for parallel processing. Specifically, the runtime systems do not partition the work (computations and data) of parallel applications across CPUs and GPUs to increase their utilization. For that reason, many existing frameworks have runtime systems that enable either static or dynamic task partitioning [5, 9–13]. However, those frameworks either fail to provide high-level abstractions, support only multi-GPU systems, or do not partition tasks to both CPU and GPU simultaneously. The observations above prompt for systems that can both exploit task partitioning efficiently and provide high-level abstractions for CPU-GPU programming.

In this paper, we propose and evaluate PSkel, a framework for stencil programming in heterogeneous CPU-GPU systems. PSkel provides a single high-level abstraction programming across CPU and GPU, and an API so the programmer can choose how to partition tasks and assign data and computation to both CPU and GPU. PSkel targets the Intel Threading Building Blocks (TBB) and the NVIDIA CUDA platforms. TBB’s work-stealing thread execution...
model can balance the parallel workload of stencil computations and provides higher performance scalability than models like OpenACC and OpenMP. However, PSkel can be easily extended to accommodate those programming models. We evaluate the performance of PSkel using five stencil applications. For those applications, we observed that PSkel-based versions with task partitioning improves performance by up to 76% and 28%, respectively, compared to their CPU- and GPU-only versions on two different platforms.

We organize this paper as follows. Section 2 provides background on the stencil pattern and discusses related work. Section 3 introduces the PSkel framework. Section 4 describes the case study applications. Finally, Sections 5 through 6 present our evaluation methodology, results, and conclusion.

2 Background And Related Work

In this section, we contextualize the main characteristics of PSkel. Specifically, we introduce the concept of stencil applications and describe past efforts in providing high-level GPU programming and task partitioning in CPU-GPU systems.

Parallel skeletons and the stencil pattern. A structured parallel program is typically composed of computation and coordination. Computation represents the application’s logic and data flow control, whereas coordination manages parallelism and concurrency. Coordination concerns include process and thread synchronization, communication, and load-balancing [14]. Since several parallel applications exhibit common computation and coordination patterns, programmers often model those patterns as parallel skeletons. With skeletons the programmer can focus on designing parallel algorithms rather than worrying about runtime system details, thereby speeding up application development and debugging. Moreover, skeletons add structure to parallel programs, so programmers can build more complex skeletons from simpler ones. Among the several existing patterns of parallel skeletons (e.g., map, reduce, pipeline), the stencil pattern has been used in applications of many important fields, such as quantum physics, weather forecasting, and digital image processing. Due to its importance, many recent efforts in GPU research sought to improve the performance of stencil computations [2, 4, 5, 7, 15–18].

The stencil pattern operates on n-dimensional data structures, using an input data value and its neighbors to compute the corresponding output data element. Specifically, a sliding window (or mask) scans the entire input data set and produces output data using a stencil function. The mask size corresponds to a specific number of neighbors of each element of the input data. The stencil function performs computations using the mask and the neighbors of each element of the input data to produce a corresponding element in the output data. The stencil application repeats that process on every element of the input data, except those at its limits (e.g., border pixels of a 2-D image) [3].

High-level GPU and CPU-GPU programming. There has been extensive work in providing high-level abstractions to make GPU programming easier. For example, previous works propose data-structure and control-flow abstractions via annotated code, compiler directives, code unification across CPU and GPU, new and extended programming languages. In this section, we focus primarily on the approaches that use parallel skeletons.

Typically, skeleton-based approaches not only add high-level abstractions to GPU-based systems, but also produce portable code and enable the tuning of the runtime system, hardware, and/or application to improve overall workload performance [14]. Kamil et al. [16] propose a framework that automatically parallelizes Fortran 95 stencil kernels into tuned Fortran, C, or CUDA with substantial gains over the sequential implementations. Emygren & Kessler [6] propose a template library in C++ that manipulates vectors and implements several skeletons (map, reduce, map reduce, map overlap, map array, and scan) for single-GPU (CUDA/OpenCL code) or CPU (OpenMP/sequential) execution. In a similar approach, Steuwer et al. propose an OpenCL-based library for multi-GPU programming [8]. Their library manipulates input and output data as vectors, performs implicit data exchange between main memory and GPU memory, and provides abstractions for four parallel skeletons (map, zip, reduce, and scan).

Like PSkel, the approach by Christen et al. generates code and provides abstractions for stencil programs on CPU-GPU systems [5]. However, their approach requires that stencil kernels are described in a domain-specific language. Holewinski et al. [4] modify the compiler to automatically generate GPU code with overlapped-tiling, which trades computational effort for reduced memory bandwidth. In a different approach, the framework by Lutz et al. [7] generates optimized OpenCL code and automatically distributes stencil computations across GPUs based on characteristics of the stencil function, problem size, GPU heterogeneity, and PCI-express settings.

Finally, the OpenACC standard for parallel computing was proposed to simplify parallel programming of heterogeneous CPU-GPU systems [19]. However, unlike PSkel, the current OpenACC API and platform cannot partition work across CPU and GPU and utilize both processing units simultaneously.

Task partitioning on GPU-only and CPU-GPU systems. Several past efforts focus task partitioning in GPU and
CPU-GPU systems. Ravi et al. [11] propose a compiler and runtime framework that map annotated reduction patterns (in C) to CPU-GPU systems. Grewe & O’Boyle [9] improve on their results by adopting a ML-based compiler model that performs static partitioning of data-parallel tasks. Unlike PSkel, their approach produces OpenCL code, and does not focus on abstractions for the stencil pattern. Kim et al. [10] take a different path and perform homogeneous task partitioning across devices in multi-GPU systems. Their approach assumes homogeneous device performance, requiring no training phase. Also pursuing the elimination of training, Boyer et al. [20] progressively assigns data and computation chunks of adaptable sizes to GPU devices based on their dynamic availability and utilization levels. Teodoro et al. [12] propose an event-driven framework for data-flow-based filtering in heterogeneous cluster systems. Their framework can use either CPU or GPU for event processing, based on a pre-defined decision policy. Wang & Ren [13] propose a method of work distribution and frequency scaling of the processing units in CPU-GPU systems to optimize energy efficiency. The approach entails exploring a large parameter space and targets reduction applications, which cannot be generalized for other patterns.

Some approaches feature low-level optimizations without high-level abstractions and/or without generalizing for one or more classes of parallel skeletons. For example, the framework by Joselli et al. [21] enables the custom heuristics for the distribution of physics computations. Kim et al. [22] leverage the heterogeneity of CPUs and GPUs and a lock-free work stealing policy to exploit specifics of their collision detection application. Cederman & Tsigas [23] compare load-balancing methods for multi-GPU systems (e.g., centralized non-blocking task queue and task stealing).

Finally, in literature, we find many frameworks for programming in GPU-based clusters [24, 25], which is not our current focus with PSkel.

### 3 The PSkel Framework

In this section, we describe the PSkel framework in light of its two main goals. First, PSkel seeks to provide a single high-level abstraction for stencil programming on heterogeneous CPU-GPU systems. Second, PSkel seeks to exploit the parallelism of those heterogeneous systems by assigning fractions of the total work to CPU, GPU, or both, as defined by the programmer.

To reach our first goal, we provide an application programming interface (API) that leverages the extensibility of C++ to provide common stencil functionality. Using parallel skeletons, PSkel releases the programmer from the responsibility of writing boiler-plate code for stencil programming (e.g., explicit synchronization and data exchanges between GPU memory and main memory). Furthermore, the framework translates the abstractions described using its API into low-level C++ code compatible with both Intel TBB and NVIDIA CUDA. To reach our second goal, we provide compile-time static task partitioning in PSkel. In particular, in generating low-level code, PSkel partitions data across GPU and CPU based on a programmer-defined parameter.

#### Application Programming Interface.

At its heart, PSkel’s API is a C++ template library that implements a stencil parallel skeleton and provides useful constructs for developing parallel stencil applications. The API provides templates for manipulating input and output data; specifying stencil masks; encapsulating memory management, computations, and runtime details.

Before delving into the individual components of the API, we summarize the general process of developing stencil applications with PSkel. In that process, some typical programmer tasks include:

1. Identifying the problem’s dimensionality (e.g., 1D, 2D, 3D); defining suitable data types for input, output, and mask arrays; and instantiating those data structures using PSkel’s data container constructs;
2. Defining a stencilKernel method that describes the computation of a single data element of the output array based on elements of the input array and mask;
3. Instantiating one or more Stencil objects, which manage memory allocation, encapsulate computations, and perform calls to the stencilKernel method to execute the computations;
4. Instantiating one or more Runtime objects, which encapsulate the handling of Stencil objects, provide an abstraction layer to the underlying low-level implementation (currently, CUDA and TBB), and orchestrate load distribution across CPU and GPU.

Each task is associated with a specific API element of PSkel, which we describe next.
3.1 Data Containers

PSkel’s API provides read/write and read-only data containers. In the current implementation of PSkel, the available read/write data containers are arrays to store input and output data for the stencil computation; and the read-only containers are stencil masks and custom arguments.

The array-based data containers transparently handle memory allocation on both GPU and CPU memories. The allocation is automatically optimized with respect to the size of the data and the sizes of the task partitions.

Input and output arrays. Because the dimensionality of inputs and outputs may vary across applications, PSkel provides three default template classes as data containers: Array, Array2D, and Array3D, respectively for one-, two-, and three-dimensional stencils. We illustrate the instantiation and initialization of PSkel data containers in Code 1. Lines 1–3 exemplify how to instantiate a PSkel array, write to and read from its elements.

Arguments. Arguments store application-specific parameters of arbitrary nature that can be passed into the core of the stencil computation (e.g., coefficients) by an instance of the Stencil class (explained below). To address the need of read-only arrays, PSkel provides three template classes: Args, Args2D, and Args3D respectively for one-, two-, and three-dimensional arrays. In the current version of PSkel, an Argument can be an instance of a custom C++ structure (composed of read-only arrays and/or native type variables) or a single native type variable/read-only array. We illustrate the use of PSkel Arguments in Code 4.

Mask. Stencil applications may have different requirements for stencil masks. For example, when calculating one element of the output array, some applications use a dense stencil mask, where mask elements match all neighbors within a grid around the corresponding input element. Conversely, other applications use a sparse stencil mask that is only applied to a few selected neighbors with fixed position relative to the current center element.

Based on that observation, by default, PSkel provides a flexible API for defining stencil masks. PSkel stores mask weights into a unidimensional array, independently of the actual dimensionality of the mask, and stores the position of each weight relative to the central element of the stencil. This implementation decision favors large sparse masks over small dense ones.

By default, PSkel provides 1D, 2D, and 3D mask abstractions. To create a Mask object, the programmer first instantiates the object, informing the total number of elements in the mask (Code 2, line 1), then initializes each element using the set method. The method takes as arguments the index of the linear array that will be associated with the element; the relative coordinates of the mask element (relative to the central mask element); and the weight of the element (if not informed, the default is 1). For example, in Code 2, line 2, the index 0 of the 3D mask is associated with the relative position \((-1, 0, 1)\) and the weight assigned to that mask element is 22.3.

PSkel provides two methods for accessing the data in the mask elements: getWeight and get. As shown in Code 2, line 3, getWeight returns the weight stored in the mask element, given its linear index. The get method is a utility method for abstracting offset calculations involved in matching mask elements to neighbor elements. Given a reference element in the input array, get returns a neighbor in the same array that maps to a given mask element. For example, in Code 2, line 4, the reference number is given by point \((10, 20, 30)\) in the input array. In that case, get applies the relative coordinate of element 0 (previously set as \((-1, 0, 1)\)) to the coordinates of the reference point, thus returning the value of the neighbor at coordinates \((9, 20, 31)\).

Code 1: Example of 3D Array in PSkel

```
1 # Array3D<float> arr_3d(10,5,20) // instantiating 3D array of size 10x5x20
2 arr_3d(0,0,4) = 10.5;     // writing a value to an array element
3 float a = arr_3d(0,0,4);  // reading the value back from the array
```

Code 2: Example of 3D Mask in PSkel

```
1 Mask3D<float> mask(6);  // instantiates sparse matrix with 6 elements
2 mask.set(0,-1,0,1,22.3); // set the position and weight of element 0
3 float mask_wgt = mask.getWeight(0); // read mask weight of element 0
4 float neig_val = mask.get(0,input,10,20,30)); // read neighbor value
```
3.2 Application’s Core Logic

The stencilKernel method. To implement the core logic of the stencil application in PSkel, the programmer must provide an implementation of the stencilKernel method. The method may use native C++ and local temporary variables within its body, but must operate on PSkel data containers for input and output. This design methodology ensures that the stencil kernel can be invoked by the different underlying platforms transparently and enables the programmer to focus thoroughly on the stencil development.

In essence, stencilKernel implements the logic to process a single element of the output array, provided access to the input array, mask, arguments, and the location of the input element currently being processed. Internally, PSkel’s Runtime system assigns a stencilKernel execution to a particular CPU or GPU processing element, and provides the necessary and relevant arguments to the method at execution time.

As an example, Code 3 shows the prototype of a typical implementation of stencilKernel. In the example, the physical CPU or GPU processing element that will execute the 2D stencil code gets access to the input and output arrays, mask, application-specific arguments, and the position \((i, j)\) of the target output element. The current implementation of the framework provides callbacks for 1D, 2D, and 3D stencilKernel functions and the Argument data containers are optional.

```c++
__stencil__ void stencilKernel(Array2D<int> input, Array2D<int> output,
2     Mask2D<int> mask, Arguments args, int i, int j);
```

Code 3: Example of a 2D stencilKernel prototype.

3.3 Stencil Configuration

When writing a stencil application in CUDA, typical programmer’s concerns include explicit memory allocation in the GPU, data copying from the host memory to the GPU’s memory, block and grid size specification, writing the program’s kernel, and only then calling the kernel. In addition, the programmer has to reason about how to distribute data and computations across CUDA threads. The complexity of this task increases significantly if the programmer decides to subdivide the computation across CPU and GPU threads.

To address those complexities, PSkel provides two types of high-level abstractions that cover most details of memory management, program execution, and task partitioning. Specifically, PSkel provides the Stencil and Runtime template classes.

Stencil template classes. The Stencil class follows the Command design pattern, and carries the configurations necessary to run a stencil kernel, i.e., the references to the necessary data containers. As shown in Code 4, the Stencil constructor requires references to an input data container, an output data container, a mask, and optional arguments. The current version of PSkel includes three derived classes that implement the aforementioned features optimized for 1D, 2D, and 3D stencil computations.

Internally, Stencil encapsulates GPU memory management (allocation, deallocation, and copying to/from the GPU memory), block and grid sizing, and is responsible for binding the calls to stencilKernel to the correct processing elements of the CPU and/or GPU.

Stencil objects are used as transient state for the actual stencil execution performed by instances of the Runtime class. As so, the contents of Stencil instances, as well as their data containers may not be preserved throughout and after the execution of the stencil computations.

Applications with multiple kernels. Stencil applications may have more than one kernel. For example, certain three-dimensional problems may be broken down into three stencil kernels (one per dimension), in which case the application can compute the kernels commutatively. To address that scenario in PSkel, the programmer simply instantiates one Stencil object for each kernel.

Grid and block sizing. The Stencil class provides a parametrized selection of GPU grid size as a function of the block size. By default, the block size is set as equal to the warp size. However, the programmer can specify custom block sizes and pass them down to the Stencil instances via Runtime objects. In most recent CUDA implementations (6.5 and above), the block size can be automatically set by the cudaOccupancyMaxPotentialBlockSize function, which predicts the block size aiming to maximize the occupancy of GPU cores.
4 Case Study Applications

3.4 Stencil Execution

Runtime Template Class. The Runtime class follows the Facade design pattern and abstracts several low-level details of stencil execution, given the configurations provided in the instances of Stencil. In particular, the Runtime class abstracts the data partitioning between CPU and GPU, the boiler-plate code for data transfers, and the execution of the user-defined stencil computations.

The programmer instantiates one Runtime object per Stencil object, in order to keep track of the execution progress of that stencil kernel. The dimensionality and data types of the Runtime and the Stencil instances must match, otherwise the programmer will get a compile-time error. As shown in Code 5, when instantiating a Runtime object, the programmer must pass a reference to a Stencil object (line 1), which will be modified and consumed throughout the execution of the application.

The Runtime class provides the method run for executing stencils while enabling task partitioning across CPU and GPU threads. As shown in Code 5, line 2, the method takes the fraction of the input data that will be processed by GPU threads (gfrac), the GPU block size (bsize), and the number of CPU threads involved in the the computation (cthreads). If the gfrac is set to 0%, bsize is ignored, and if gfrac is set to 100%, cthreads is ignored.

Task partitioning. Given the gfrac argument of the run method, Runtime objects partition 2D input and output data containers into contiguous sets of rows. The fraction of the rows given by gfrac is assigned to the GPU and subdivided across its threads, and the complement of the fraction is assigned to CPU threads. Similarly, for 1D containers, Runtime assigns fractions of the linear array to CPU and GPU; and for 3D containers, Runtime assigns contiguous three-dimensional matrices to each type of processing element.

Iterative stencil applications. Stencil applications may be iterative, i.e, the output of a full step of stencil computation (iteration) can be used as input for the next iteration, and the compounded output is meaningful or useful. In the presence of task partitioning, at the end of an iteration, the application has a CPU-processed buffer and a GPU-processed buffer. However, to proceed to the next iteration, data on the adjacent borders of both buffers must be updated, and therefore exchanged across both buffers.

PSkel performs that border exchange automatically at the end of each iteration by the instances of the Runtime class. To execute the application iteratively, the programmer replaces run with runIterator, which takes similar arguments, plus an additional argument that specifies the number of iterations (ites) of the program (Code 5, line 3).

4 Case Study Applications

In this section, we describe five stencil applications that we use to evaluate PSkel. The applications cover a wide range of stencil computations, from 2D to 3D inputs, iterative and non-iterative behavior, balanced and unbalanced tasks. We
also show our implementation of their stencil kernel functions to illustrate the use of PSkel.

**Convolution.** Image convolution is a technique used in many fields of study. In digital image processing, for example, we can use convolution to implement frequency filtering applications that smooth or highlight the input images. Code 6 shows the PSkel stencil function for our image convolution application, to which we refer as Convolution, for simplicity.

Convolution processes a $W \times H$ input image $I$ into an output image $O$ of same dimensions, using a square coefficient matrix $M$ of side $C$. For a pixel $I_{i,j}$ in $I$ (0 < $i < W$ and 0 < $j < H$), our application computes the corresponding pixel $O_{i,j}$ in $O$. We compute $O_{i,j}$ as a summation of the multiplication of each surrounding neighbor of $I_{i,j}$ by a numerical coefficient in $M_{l,k}$ in $M$ ($l < C$ and $k < C$). In the computation of the pixels on the border of $I$, we ignore the neighboring pixels that lie outside of the boundaries of $I$ (e.g., the pixel above the top left pixel of $I$) [26].

```c
__stencil__ void stencilKernel(Array2D<float> input, Array2D<float> output,
                              Mask2D<float> mask, int i, int j)
{
    float accum = 0.0;
    for(int n=0; n<mask.size; n++)
        accum += mask.get(n,input,i,j) * mask.getWeight(n);
    output(i,j) = accum;
}
```

**Code 6:** Stencil kernel for Convolution in PSkel

**FAST (corner detection).** Computer vision programs typically use corner detection to extract certain features from an image. For example, those features enable the program to infer image content, detect motion, create 3D models, and recognize objects [18].

In our evaluation, we adopt the FAST (Features from Accelerated Segment Test) algorithm for corner detection [17, 18]. The FAST algorithm considers that a pixel $p$ is corner if, in a 16-pixel circle around $p$ (Bresenham circle of radius 3), at least 9 consecutive pixels are all above or below the value of $p$ by a threshold $t$. Code 7 shows the PSkel stencil function of our corner detection application, referred to as FAST, for simplicity. To implement the FAST stencil function, we first define a mask as a matrix containing the positions relative to $p$ that must be analyzed. The mask scans the relative positions and verifies whether 9 consecutive pixels satisfy the corner detection condition. The application detects all corners in the input image.

**GoL (Game of Life).** “Game of Life” is a cellular automaton implementing Conway’s Game of Life [27]. We implement the automaton as a matrix in which each cell represents a living or a dead individual. Over the course of a pre-defined number of iterations (or “generations”), each individual analyzes the state of its neighbors to define its own state in the next iteration. There are four possible neighborhood conditions that define the next state of a cell. First, a dead cell with exactly three living neighbors becomes alive at the next time step, by “reproduction”. Second, a living cell with less than two living neighbors dies, due “loneliness”. Third, a living cell with more than three living neighbors dies, due “insufficient life-supporting resources”. Forth, a living cell with two or three living neighbors or a dead cell with more or less than three neighbors maintains its previous state.

GoL has been used in several studies since its proposal, and it is known to consume significant amounts of computing resources. The GoL computations follow the stencil pattern, i.e., we can use a mask to select multiple neighboring elements from the input matrix and compute the state of the center element, then repeat that step for all elements in the input. In Code 8, we show our implementation of the stencil computations of GoL. For brevity, we do not show the complete automaton code.

**Laplacian.** The Laplace operator ($\Delta$), or Laplacian, is used in differential equations that describe many physical phenomena, such as electric potential, heat diffusion, and wave propagation. When applied to a function, the operator denotes the divergence of the function’s gradient on Euclidean space (Equation 1), and is equivalent to the sum of the unmixed second partial derivatives at point $x_i$. Our stencil application implements the second-order finite difference discretization of the Laplacian in 3D space (Equation 2). For each element $e$ in the input matrix, the stencil computes the rate at which the average value of the neighbors deviates $e$. That computation can be parametrized by the weights $\alpha$ and $\beta$. The resulting stencil kernel is shown in Code 9.

\[ \Delta := \sum_i \frac{\partial^2}{\partial x_i^2} \]  
\[ u'_{ijk} = \alpha u_{ijk} + \beta (u_{i\pm1,j,k} + u_{i,j\pm1,k} + u_{i,j,k\pm1}) \]
5 Experimental Results

In this section, we evaluate PSkel and compare it to different baseline systems, using the applications described in Section 4. First, we explain our evaluation methodology and then we present and discuss our results.

Gauss-Seidel with Red-Black parallelization (GSRB). GSRB is a high-order discretization of the 3D Laplace operator, capable of exploiting parallelism more deeply than the previous approach. The method partitions the input matrix into interleaved parts that can be processed independently. The element-wise Laplacian operation in GSRB is given by Equation 3, where $i, j \in \mathbb{Z}^3$, $j$ is an offset vector, and $a, b, c_1,$ and $c_2$ are parameters expressed as functions of the weights $\alpha$ and $\beta$. As an optimization, Christen et al. collapse two iterations into one [28], by setting $a = \alpha + 6 \beta^2$, $b = \alpha \beta$, $c_1 = 2 \beta^2$, and $c_2 = \beta^2$. We show the resulting computation as a PSkel stencil kernel in Code 10.

\[
    u'_i = au_i + b \sum_{|j_1|=1} u_{i+j_1} + \sum_{r=1}^{2} c_r \sum_{|j_1|=2,j_2 \in \{0,\pm r\},k=1,2,3} u_{i+j_1+j_2}
\]

Code 7: Stencil kernel for FAST in PSkel

```c
__stencil__ void stencilKernel(Array2D<int> input, Array2D<int> output,
Mask2D<float> mask, int T, int i, int j){
    int accumBrighter, accumDarker, int imagePixel;
    int centralPixel = input(i,j);
    for(int z=0; z<16; z++){
        accumBrighter = 0;
        accumDarker = 0;
        for(int r=0; r<N; r++){
            imagePixel = mask.get(r+z,input,i,j);
            if(imagePixel >= (centralPixel + T)){
                if (accumBrighter == 0){
                    accumDarker++;
                    z += r - 1;
                    continue;
                }
            } else if(imagePixel <= (centralPixel - T)){
                if (accumDarker == 0){
                    accumBrighter++;
                    z += r - 1;
                    continue;
                }
            } else{
                z += r;
                continue;
            }
        }
        if((accumBrighter == N || accumDarker == N)){
            output(i,j) = 300;
            z = 16;
        }
    }
}
```

Code 8: Stencil kernel for GoL in PSkel

```c
__stencil__ void stencilKernel(Array2D<int> input, Array2D<int> output,
Mask2D<int> mask, int i, int j){
    int neigh=0; // neighbors
    for(int z=0; z<mask.size; z++)
        neigh += mask.get(z,input,i,j);
    output(i,j) = (neigh==3 || (input(i,j)==1 && neigh==2)) ? 1 : 0;
}
```
5.1 Evaluation Method

Application settings. The stencil applications that we consider in our experiments use different input values and configurations. In Convolution and FAST, we process a set of high-resolution (HD standard) input images, namely, images of densities: 2.1 megapixels (1920x1080 array), 8.3 megapixels (3840x2160 array), and 33.1 megapixels (7680x4320 array). Convolution uses a 5x5 dense mask, whereas FAST uses a 16-neighbor sparse mask. In GoL, we process 2D arrays that were randomly generated to represent the initial state of 8000x8000 worlds. We process the array using an 8-element mask (3x3 without the central element). In Laplacian and GSRB, we process a 3D input array with dimensions 480x480x480, respectively using sparse masks of size 7 and 25 neighbors. Because GoL, Laplacian, and GSRB are iterative applications, we evaluate their performance for different number of iterations.

Baseline versions. To evaluate PSkel, we compare the performance of several versions of each aforementioned application. In particular, we compare:

- Sequential: a C++ sequential implementation of the application running on a single CPU core;
- CPU-only: a parallel implementation running on the CPU alone, but running on as many CPU cores as possible;
- GPU-only: a parallel implementation, running on the GPU alone and utilizing as many GPU cores as possible;
- PSkel Best Static Partition (PSkelBSP): the parallel implementation with task partitioning that exhibits the best average performance on the largest input sizes. To find PSkelBSP for a given application on a particular platform, we ran experiments using the largest available inputs. In each experiment, we assigned a percentage of work to the GPU (10% to 90%, in increments of 10%) and measured the response time. PSkelBSP is the configuration exhibiting the lowest average execution time. We use the notation GPU-\(p\)% to denote a task partition scheme that assigns \(p\)% work to the GPU and \((1 - p)\)% to the CPU.

Note that GPU-0% and GPU-100% are PSkel based implementations rather than pure TBB and CUDA implementations, respectively. However, in those extreme cases PSkel works as a wrapper and imposed insignificant overhead in our experiments. For that reason, we do not explicitly report results for pure TBB and pure CUDA implementations.
In addition, notice that our goal is to explore and showcase the potential benefits of task partitioning, rather than claiming that a particular approach is better than all others. In future work, however, we intend to exploit PSkel’s ability to partition work to automatically select the best partition for given application, input size, and platform combination.

**Performance metric.** To compare the different parallel approaches in each case study, we measured their response time and computed their speedup relative to Sequential. We report the geometric mean of that speedup across ten executions of each experiment.

**Evaluation platforms.** We evaluated PSkel on two CPU-GPU platforms. Table 1 shows the features of the systems, both running on Ubuntu Linux 12.10 with CUDA 5.5 (driver v331.13) and TBB from the Ubuntu repository (libtbb-dev 4.0+r233-1).

<table>
<thead>
<tr>
<th>Feature</th>
<th>Platform 1</th>
<th>Platform 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Model: Intel Core i7 3610QM; Frequency: 2.3GHz; Physical cores: 4; Logical cores: 8; Last level cache: 6MiB</td>
<td>Model: AMD FX-8120; Frequency: 3.1GHz; Physical cores: 8; Logical cores: 8; Last level cache: 8MiB</td>
</tr>
<tr>
<td>Memory</td>
<td>Capacity: 8GiB; Bus: 1600 MHz DDR3</td>
<td>Capacity: 32GiB; Bus: 1333 MHz DDR3</td>
</tr>
<tr>
<td>GPU</td>
<td>Model: GeForce GT 630M; Cores: 96; Core frequency: 660MHz; Memory capacity: 2GiB; Memory bus: 900 MHz DDR3</td>
<td>Model: GeForce GTX 590; Cores: 512; Core frequency: 630MHz; Memory capacity: 1.5GiB; Memory bus: 864 MHz GDDR5</td>
</tr>
</tbody>
</table>

5.2 Results and Discussion

Figure 1 shows the speedup of the parallel implementations of Convolution (i.e., CPU-only, GPU-only, and PSkelBSP) over Sequential. In particular, Figures 1(a) and 1(b), depict that comparison on platforms 1 and 2, respectively. On platform 1, PSkelBSP entails assigning 70% of the task to the GPU, whereas on platform 2, the assigned fraction is 60%. Because Convolution is embarrassingly parallel and exhibits homogeneous processing time per input array element, the application’s performance is heavily influenced by the processing power and parallelism that the processing units can deliver. For example, on platform 1, the GPU has significantly more computing power than the CPU, hence GPU-only consistently dominates CPU-only. Interestingly, on platform 2, the GPU also has significantly more computing power than the CPU, however CPU-only and GPU-only exhibit similar performance. The reason is that at full utilization both processing elements perform well for the input sizes that we considered, and GPU-only has a more significant advantage when processing the largest input. Regardless, on both platforms, we observe that harnessing the joint computing power of CPU and GPU can improve performance significantly. In fact, PSkelBSP improves the average performance of Convolution across all platforms by 34% over CPU-only (maximum of 60%) and 16% over GPU-only (maximum of 23%).

Figures 1(c) and 1(d) show the speedup of the parallel FAST implementations over Sequential, when executed on platforms 1 and 2, respectively. Unlike in Convolution, the processing time per element in FAST is fairly uneven. In fact, for smaller input sizes, CPU-only dominates the other approaches, because in addition to parallelizing the application, TBB uses work stealing to perform load balancing. By balancing load across hardware threads, TBB increases the CPU’s utilization, whereas CUDA does little to leverage the idle processing elements at a given time. However, as the input size increases, the GPU processing elements become increasingly utilized, therefore the relative impact of load balancing in the CPU diminishes. For similar reasons, PSkelBSP performs better than CPU-only for larger inputs and always performs better than GPU-only. Across all platforms and workloads that we tested, on average, PSkelBSP performs within 2% of CPU-only and 15% better than GPU-only. However, for the large input sizes, PSkelBSP improves the average performance by 42% (maximum of 53%) compared to CPU-only and by 21% (maximum of 28%) compared to GPU-only.

Figure 2 shows the speedup of CPU-only, GPU-only, and PSkelBSP compared to Sequential three iterative applications. For each application, namely GoL, Laplacian, and GSRB, we varied the number of iterations from 1 through 5 and measured execution times. In our experiments, the first iteration entailed copying the input data into a temporary buffer that was CPU-accessible (via memcpy) and/or to the internal memory of the GPU, in the cases where the GPU was used. The subsequent iterations required fewer data transfers because platforms 1 and 2 had enough memory to store the input and output arrays in each experiment.
Experimental Results

As expected, reducing the amount of data transfers improved application performance in almost all scenarios for GPU-only and PSkelBSP. We observe that in all cases, CPU-only does not scale beyond the total number of cores in either platform. In addition, CPU-only exhibited a steady speedup, with little variance after the second iteration, because Sequential also observed slight performance improvement in the subsequent iterations. On average, across all platforms, PSkelBSP performed 35%, 24%, and 69% better than CPU-only, respectively for GoL, Laplacian, and GSRB, whereas GPU-only performed 45%, 37%, and 74% better than CPU-only for the same applications.

Consistently with the results for the non-iterative applications, on platform 1, PSkelBSP exhibited better average performance than GPU-only by 10% (maximum of 14%), 10% (maximum of 13%), and 16% (maximum of 18%), respectively in GoL, Laplacian and GSRB. In contrast, in platform 2, GPU-only performed far better than the other approaches. The reason is that in that platform, the GPU has significantly more processing elements than its CPU, thus, assigning larger work fractions to the GPU across multiple iterations translated directly into higher speedup. Notably, GSRB attains very high speedups on platform 2, due to the high parallelism of the application. On that platform, on average, we observed that, GPU-only performed better than PSkelBSP by 29%, 29%, and 42%, respectively in GoL, Laplacian and GSRB.

To further investigate the disparity of results across platforms, we evaluated different aspects of the applications and noticed that they differed significantly in data transfer time. In Figure 3, we break down execution time of GSRB into processing time, CPU-to-CPU data transfer time (cpu2cpu) and CPU-to/from-GPU data transfer time (cpu2gpu). Processing time is the time spent in useful computations towards the final output of the stencil computation. CPU-to-CPU is the sum of the time involved in copying data across CPU-accessible temporary buffers (e.g., input array into temporary buffer for processing). GPU-to/from-CPU comprises the time spent in border exchanges between CPU-processed and GPU-processed data buffers (as described in Section 3).

Figure 3 reveals that on platform 2, a non-trivial percentage of the execution time is spent on data transfers, as the fraction of work assigned to the GPU increases. However, because the GPU is significantly more powerful than the CPU, the time spent in data transfers in GPU-10% through GPU-90% offset most benefits of partitioning, as...
compared to GPU-only. We observe that in PSkelBSP (GPU-90%, in this case), approximately 60% of the execution time is due to both CPU-to-CPU and CPU-to/from-GPU data transfers, which significantly decrease the performance of PSkelBSP. Conversely, on platform 1, the time consumed in data transferred is insignificant, thus the performance difference between CPU and GPU impacts application performance more significantly. In that scenario, across all iterative applications, PSkelBSP outperforms CPU-only by 61% on average (maximum of 68% for GSRB) and GPU-only by 12% (maximum of 15% for GoL).

From the results above, we conclude that harnessing the processing power of both CPU and GPU can improve performance significantly for the parallel applications that we analyzed. In our experiments, we noticed that a parallel implementation that partitions work was able to improve performance by as much as 76% compared to a GPU-only version and by 28% compared to a CPU-only version. In that context, having a framework like PSkel not only provides the means to automatically perform task partitioning for stencil applications, but also abstracts GPU boiler-plate code and adds an extensible, unified, and consistent C++ API across GPU and CPU.
6 Conclusion

In this work, we presented PSkel, a stencil programming framework for CPU-GPU systems. PSkel is based on parallel skeletons and provides a high-level programming interface for GPU stencil programming, while enabling the programmer to partition work across CPU and GPU.

Our results indicate that task partitioning may improve the overall application performance in many cases, as compared to using only the CPU or only the GPU. Since optimal partitions may vary on a case-by-case basis, by enabling task partitioning, PSkel creates opportunity for investigating and comparing different task partitioning approaches. We also show that high-level programming interfaces can exploit the high performance of GPUs, while improving their ease of programming.

Finally, the framework’s API is extensible and allows for different optimizations, task partitioning policies, and underlying implementations. PSkel is ongoing work and our future research includes: (1) expanding the framework with new parallel programming patterns; (2) studying automatic optimizations and adaptive optimization algorithms; and (3) including support to other parallel platforms (e.g., AMD GPUs and Intel Xeon Phi coprocessors) by adding new back-ends to the framework, specifically using the OpenCL, OpenACC, and/or OpenMP programming models.

References


