Hybrid CPU-GPU Pipeline Framework PDPTA'14

Fahad Khalid*, Frank Feinbube, Andreas Polze
Hasso Plattner Institute for Software Systems Engineering
14482 Potsdam, Germany
fahad.khalid, frank.feinbube, andreas.polze@hpi.uni-potsdam.de

Abstract— *The pipeline pattern for parallel programs is utilized in a wide array of scientific applications designed for execution on hybrid CPU-GPU architectures. However, there is a dearth of tools and libraries to support implementation of pipeline parallelism for hybrid architectures.*
We present the Hybrid Pipeline Framework (HyPi) that is intended to fill this gap. HyPi provides high level abstractions in C++ for implementation of pipelines on hybrid CPU-GPU architectures. It is a generic framework intended to support a wide range of applications. The complexities characteristic of such implementations, e.g., partitioning of input/output data structures, asynchronous memory transfer, communication between CPU and GPU etc., are handled by the framework and are therefore hidden from the developer. HyPi exposes certain degrees of freedom that can be tuned to optimize the performance of a simulation based on application specific requirements. We present a detailed account of the framework design, and evaluate the framework performance using a real-world application from the domain of computational biology. Results show that HyPi performs on par with a custom-tailored, hand-tuned pipeline implementation for the given application.

Keywords: Hybrid pipeline, heterogeneous computing, pipeline parallelism, Intel Threading Building Blocks, CUDA

1. Introduction

The advent of the CUDA programming model marked the emergence of mainstream application of accelerator programming to scientific computing. Over the years, the said programming model has evolved significantly; supporting a substantial number of advanced features. A large number of algorithms have been ported to the GPU architecture; several of which are available in CUDA based libraries [1], [2]. Yet, even today, programming a GPU costs much more in terms of productivity than programming a CPU for the same problem. Therefore, even though applications capable of execution on a GPU may benefit in terms of speedup, the effort required to engineer such applications reduces programmer productivity. In short, the decision of whether or not to port an application to the GPU must balance the trade-off between application performance and developer productivity. We term this trade-off the *productivity vs. performance trade-off*.

The *pipeline* pattern [3], [4] for parallel programming covers a broad range of applications for which the *productivity vs. performance trade-off* is evident. A commonly recurring application in hybrid CPU-GPU architectures is the overlap of computation and transfer of memory from GPU to CPU (and vice versa). Other applications include the execution of different stages of a program on either CPU or GPU, in order to properly utilize all processing resources [5], [6].

Even though the CUDA programming model provides features that make it possible to implement a cross-device pipeline, e.g., using CUDA *streams* for asynchronous memory transfer, the implementation details render the process rather cumbersome. It is conceivable that if the process of implementing cross-device pipelines is simplified, researchers will be incentivized to explore the potential of implementing new algorithms using the hybrid CPU-GPU pipelining approach.

In this paper we present our *Hybrid Pipeline Framework (HyPi)* that is designed to provide high level abstractions for implementing the pipeline pattern on hybrid architectures. It is a generic framework intended to support a wide range of applications that can benefit from a hybrid pipeline. *HyPi* hides much of the intricacies inherent in implementing such a pipeline, while exposing enough degrees of freedom so that the pipeline performance can be optimized for each individual application. Moreover, we believe that *HyPi* can serve as a test bed for assessing the feasibility of implementing certain algorithms using the hybrid pipelining approach. It is important to note that *HyPi* is available as an external C++/CUDA library, and does not require any additional compiler support. The library has been tested with both GCC and the Intel C++ compiler.

The paper is organized as follows: Section 1.1 presents an overview of related work, where different applications and available frameworks for hybrid pipelining are discussed. Section 1.2 highlights the research gap and our contribution. A detailed account of framework design is presented in Section 2. Application of *HyPi* to a computational biology simulation is presented in Section 3. This is followed by comparative evaluation of *HyPi* against a custom-tailored pipeline in Section 4. Finally, Section 5 concludes the paper with a discussion of future work.

1.1 Related Work

Software pipelining has been around as a concept in computer science for a long time. A rigorous survey of various methods for software pipelining is presented in [7]. In parallel computing, pipelining has been identified as a commonly occurring pattern [8], and has therefore been the subject of study for many a research project.

A thorough literature review of the use of pipelining in hybrid computing has revealed that pipelining is commonly utilized in three different situations: 1) Overlapping computation and transfer of data between CPU memory and GPU memory; 2) execution of different stages of a program on either CPU or GPU based on which architecture is better suited to the computation; and, 3) execution of different stages of a program on either CPU or GPU for load balancing and optimal resource utilization. Following is a selection of works that utilize pipelining in one or more of the above mentioned situations:

A *Pipelined Multi-GPU MapReduce (PMGMR)* implementation is presented in [9] where GPU acceleration is extended to multiple GPUs. In this work, the primary application of pipelining is to overlap computation and communication in order to reduce the communication overheard. The implementation also makes it possible to process datasets that exceed both CPU and GPU memory capacity.

In order to harness the power of GPU clusters for *MapReduce*, a library has been developed [10]. This work focuses on tackling the challenges of data movement between GPUs, managing out-of-core data on GPUs, as well as modifying *MapReduce* in order to leverage the GPU cluster architecture. The pipelining concept is utilized in terms of overlap of computation and communication.

The problem of efficient scheduling of *MapReduce* tasks on a coupled CPU-GPU chip is dealt with in [5]. Two different approaches are presented; 1) dynamically dividing *Map* tasks onto both CPU and GPU, and 2) pipelining *Map* and *Reduce* tasks between GPU and CPU. Empirical evidence is provided to show that a pipelining solution where *Map* is implemented on the GPU and *Reduce* is implemented on the CPU justifies the use of pipelining for *MapReduce*.

Moim [11] is a *MapReduce* framework for Multi-GPU systems that implements a 3-stage pipeline consisting of *input split*, *map* and *merge* phases. The *input split* and *merge* phases are executed on the CPU, while the *map* phase is executed on the GPU. The *reduce* phase can be executed simultaneously on GPU and CPU depending on whether or not the GPU memory can hold the entire partition to be reduced.

To summarize, in order to improve the performance of the *MapReduce* model, two types of GPU acceleration methods have been employed. In the first approach, both *Map* and *Reduce* are implemented as GPU kernels. In the second approach, *Map* is implemented on the GPU, while *Reduce* is implemented on the CPU. Justification for the second approach lies in the fact that the modern GPU architecture is particularly suitable for data parallel applications that employ the *Map* pattern [4]. In such an application, multiple instances of the *Map* function can be processed in parallel by many processing elements; thereby maximizing the utilization of processing resources on the GPU. However, a parallel implementation of the *Reduce* pattern processes data in such a way that the resource utilization (in terms of processing elements) follows a tree like pattern, i.e., the number of required processing elements (or the *degree of parallelism*) decreases over time. Therefore, utilization of the processing resources is not consistent throughout the function. Even though *dynamic parallelism* [12] can be used to improve resource utilization [13], kernel design becomes exceedingly complicated.

Hybrid CPU-GPU pipelining is not limited to *MapReduce*. A 3-stage CPU-GPU pipeline for eigenvector and eigenvalue determination is presented in [6]. The work establishes the significance of utilizing a hybrid pipeline for optimal resource utilization and presents a stochastic queue monitoring strategy for parallel applications based on the pipeline pattern.

The concept of hybrid CPU-GPU pipelining has also been applied in computational biology, where the problem of enumeration of elementary flux modes in metabolic networks was parallelized using an OpenMP and CUDA based solution [14].

The capability to implement a hybrid CPU-GPU pipeline has been introduced within the context of the FastFlow [15] framework. The FastFlow framework was extended by introducing an abstraction for creating and executing a pipeline with user-defined stages. A GPU-enabled linear algebra library can be called from within a stage, making it a GPU execution stage. Therefore, different stages of the pipeline can execute on either the CPU or the GPU.

1.2 Research Gap and Our Contribution

Most of the above mentioned related work concentrates on either utilizing pipelining for a specific programming model such as *MapReduce*, or a particular application. Therefore, most of these works lack generalizability. In contrast, the framework described in [15] is much more general and useful for a larger number of applications.

The framework we present in this paper is not limited to any specific application or a programming model such as *MapReduce*. It is a generalized framework similar to the FastFlow pipeline presented in [15], and provides features that overcome certain limitations of the FastFlow pipeline. Our *Hybrid Pipeline Framework (HyPi)* is built on the pipeline functionality provided by a widely used and robust parallel programing model, i.e., *Intel Threading Building Blocks* [16]. *HyPi* provides the following features that stand out in comparison to other frameworks:

- Automated management of CUDA *streams* and *events* for asynchronous *CPU-to-GPU* and *GPU-to-CPU* memory transfer.
- Automated management of communication between GPU and CPU using the *callback* functionality introduced in CUDA 5.0.
- The possibility of multi-threaded execution of CPU stages of the pipeline.
- Automated partitioning of input/output data structures.

In addition to presenting a detailed account of the features mentioned above, this paper provides a thorough analysis of the major issues in implementing such a generic framework with CUDA. To the best of our knowledge, no other framework provides all the features available in our *Hybrid Pipeline Framework (HyPi)*.

2. Framework Design

In the rest of the document, we refer to the CPU as *Host*, and the GPU as *Device*.

2.1 Overall Pipeline Design

One of the major *HyPi* design objectives is to provide the programmer with a familiar interface for pipeline implementation. Based on our positive experience with *Intel Threading Building Blocks (TBB)* [16], we decided to use TBB as the foundation for the pipeline. TBB is a free and open source multithreading library from Intel, which provides a reliable and efficient abstraction (*tbb::pipeline*) for pipeline implementation on multicore CPUs. Each stage of a pipeline is implemented as a C++ class that inherits *tbb::filter*.

There are two possible ways in which *HyPi* can be used to facilitate the implementation of a hybrid pipeline: 1) using *HyPi* stages, and 2) using custom-tailored stages.

When using *HyPi* stages, the programmer does not need to implement the pipeline stages. Instead, classes predefined in the framework are used. The following types of stages are pre-implemented in the framework:

- *DeviceFilter:* This class represents a pipeline stage that executes a CUDA *Device* kernel. It automatically handles partitioning of input/output data structures, asynchronous memory transfer to and from the *Device*, and *callback* mechanism required to inform the following pipeline stage of the completion of *Device-to-Host* result data transfers. Details of these features are covered in Section 2.2.
- *CallbackFilter:* This stage usually follows the *DeviceFilter* and encapsulates the *Device-to-Host* communication that takes place using *callbacks*. In TBB, a typical pipeline stage proceeds with executing its function as soon as it receives a token from the previous stage. The token generation function in *DeviceFilter* and initiation of memory transfer work asynchronously. Therefore, the stage that follows *DeviceFilter* must not just wait for the token, but also the corresponding memory transfer confirmation. The *CallbackFilter* hides this mechanism from the programmer and passes the token to the next stage only once the corresponding memory transfer is complete.
- *PostProcessFilter:* This class represents a stage that is required to receive result data from the *Device* and process it on the *Host*.

For the above mentioned automation procedure to work, the programmer is required to register the signature of the *Device* kernel to be used, and a post-processing function to be called from within the *PostProcessFilter*. This process is similar in principle to the *MapReduce* programming model where the programmer specifies that *Map* and *Reduce* functions, and the rest is taken care of by the framework.

One or more custom-tailored pipeline stages can also be provided by the programmer as C++ classes that extend *tbb::filter*. Either the entire pipeline can be constructed in this fashion, or a pipeline with *HyPi* stages can be extended to include more stages. For a pipeline that does not use any of the *HyPi* stages, *Device* programming can be simplified by using a C++/CUDA library provided in *HyPi* that exposes functions for automated data partitioning, automated kernel launch for each partition, and simplified abstractions for asynchronous data transfer from *Host* to *Device* and *Device* to *Host*. In fact, *HyPi* stages are primarily C++ classes that encapsulate these functions in an organized manner. Example of a custom-tailored pipeline initialization is presented in Figure 1.

Any stage intended for execution on the *Device*, e.g., *DeviceFilter* (as well as *CallbackFilter*), must always operate in the *serial in-order* mode. This is because parallelism is achieved by the *Device* kernel itself, and does not require multiple *Host* threads. *Host* bound stages however, can operate in *parallel* mode as well.

```
tbb::pipeline pipeline;

CallbackFilter myCbFilter;
DeviceFilter myDevFilter(... params ...,
                         myCbFilter);
PostProcessFilter myPProcFilter(... params ...,
                         myDevFilter);

numPartitions = myDevFilter.getNumPartitions();
myCbFilter.initTrasferredFlags(numPartitions);

pipeline.add_filter(myDevFilter);
pipeline.add_filter(myCbFilter);
pipeline.add_filter(myPProcFilter);

pipeline.run(numTokensInFlight);
pipeline.clear();
```

Fig. 1: A custom 3-stage pipeline initialization using *HyPi* stages (backend code).

2.2 Automation Design Challenges

The following subsections describe design considerations for each of the major features provided by *HyPi*.

2.2.1 Partitioning

In hybrid architectures, *Device* memory is generally much smaller than the *Host* memory. Given this limitation, one important design assumption in *HyPi* is that the input/output data structures must be partitioned to ensure that they fit into the *Device* memory. Since *HyPi* is a library, the information about which data structures are required as arguments by the user-defined kernel, along with the corresponding sizes of these data structures, must be provided by the programmer. Similar to OpenCL [17], each kernel argument is registered with the framework as part of the initialization code. If necessary, the size of the output data structure can be specified as a function of input size. This function must also be provided by the programmer.

This information is then used by the framework to determine the maximum partition size for each data structure. In order to determine the partition size, three factors are evaluated: 1) total *Device* memory required by all data structures, as well as the total *Device* memory available; 2) impact of partition size on pipeline performance due to memory transfer between *Host* and *Device*; and 3) efficient utilization of *page-locked* [12] *Host* memory. The user can configure factors 2 and 3 by specifying upper limits for partition size and *page-locked Host* memory. Since a small upper limit (i.e., a value which is much smaller than the total *Device* memory available) on partition size may allow multiple partitions to reside simultaneously in the *Device* memory, each set of such partitions is bundled together as a *Segment*. The segment size is controlled by the upper limit on *page-locked* memory. The use of *page-locked* memory is necessitated by the fact that the framework uses asynchronous CUDA calls for all *Host-to-Device* and *Device-to-Host* memory transfers.

Note: The first prototype supports only equal sized linear data structures for automated partitioning. However, at the time of this writing, sophisticated algorithms are being implemented to support a wider range of possibilities.

It is important to note that *page-locked Host* memory is a scarce resource. Therefore, in order to keep a good number of tokens in flight, it is necessary to employ *multiple-buffering* [18] for data structures that use *page-locked* memory. This way, processing of a single partition results in the occupation of only part of the *page-locked* memory. The next partitions in line can use the remaining portions. In the meantime, the *PostProcessFilter* can release the *page-locked* memory occupied by the first partition and make it available for use by further partitions. The *multiple-buffering* solution is implemented in *HyPi*.

2.2.2 Kernel

For *HyPi* to be able to automatically call the user-defined CUDA kernel, the kernel must be registered with the framework. As mentioned in Section 2.2.1, kernel arguments must also be registered. However, CUDA kernel configuration parameters – such as maximum grid dimension and threads per block – can be configured by the user.

2.2.3 Stream and Event Management

Once all the required information is available to the framework, for each segment, the kernel processes all partitions. Depending on the problem size, processing each partition may require launching multiple grids. For each grid, separate streams and events are used to ensure efficient asynchronous memory transfer between *Host* and *Device*. All these steps are handled by the framework.

2.2.4 Device-Host Event Communication

Once a partition has been processed by the *Device*, output generated by the kernel is transferred from *Device* to *Host* asynchronously. This means that the *Device* can start processing the next partition without waiting for the *Device-to-Host* transfer to complete. Therefore, the *DeviceFilter* may issue tokens for which the *Host* has not yet received the result data. In order to make sure that the *Host* is aware of when the data transfer is complete, *callback* functionality introduced in CUDA 5.0 is used. As soon as the data transfer is complete, a *callback* function is executed that updates the state of the *CallbackFilter* stage. This update is correlated with the token received from the *DeviceFilter*. Therefore, at this point, the *CallbackFilter* knows that the process is complete and sends out a token to the next stage.

3. Use Case

In this section, we present the application of *HyPi* to the problem of enumerating elementary flux modes in metabolic networks.

The process of metabolism comprises of chemical compounds, called *metabolites*, transformed into other chemical compounds through *reactions* catalyzed by enzymes. A group of such related *metabolites* and *reactions* can be viewed as a network, modeled mathematically as a node-weighted directed hypergraph. A node in such a hypergraph stands for the number of molecules of a particular metabolite; while a directed hyper-edge represents a reaction. *Elementary Flux Modes (EFMs)* are minimal subnetworks that operate at equilibrium. Removal of any component results in the EFM being unusable. In order to use EFMs to characterize the behavior of a metabolic system, it is required to enumerate all EFMs in the system. A commonly used algorithm for EFM enumeration is the *Nullspace* algorithm [19]. For a detailed description of the algorithm, we

refer the reader to [20]. Here we summarize the main steps of the algorithm:

1) The *Nullspace* algorithm operates on the *Stoichiometic matrix*. Rows in the stoichiometric matrix correspond to *metabolites*, and the columns correspond to *reactions*. The matrix can be viewed as the *incidence matrix* corresponding to the hypergraph that represents the metabolic network to be analyzed. The stoichiometric matrix is compressed [21] in order to improve performance.
2) An underdetermined system of homogeneous linear equations is solved to obtain the *nullspace*, where the stoichiometric matrix is the coefficient matrix. The *nullspace* is permuted to simplify further operations. The permutation results in two parts of the matrix: $R^{(1)}$ and $R^{(2)}$.
3) For each row in $R^{(2)}$:
   a) Algebraic combinations are generated for selected columns in $R^{(2)}$ and bitwise combinations are generated for the corresponding parts in $R^{(1)}$
   b) Duplicate candidates are removed
   c) Each candidate is verified for elementarity
   d) The verified candidates are appended as column vectors to the *nullspace*

A row in $R^{(2)}$ that has been processed is converted into a binary representation, and moved to $R^{(1)}$.
Due to its combinatorial nature, the candidate generation phase is extremely expensive both in terms of computation and memory. Even for small to medium sized networks, parallel computation is a necessity.

3.1 Combinatorial Candidate Generation

The *combinatorial candidate generation* algorithm refers to the generation of bitwise combinations in $R^{(1)}$. Figure 2 describes the pseudocode. $R^{(1)}$ is split into two bit matrices, **X** and **Y**. A candidate vector is generated by performing a *bitwise OR* operation between a column in **X** and a column in **Y**, and then performing a *popcount* operation on the result vector. Indices corresponding to the operand columns in **X** and **Y** are stored, marking the result vector as a candidate for elementarity testing if the *popcount* is greater than a certain threshold value λ (determined elsewhere in the *Nullspace* algorithm).

In previous work [14], we developed a hybrid pipeline based parallel solution to the combinatorial candidate generation problem. There we implemented a pipeline using OpenMP and CUDA that was tightly coupled with, and optimized for the given problem. In our current work, we present an implementation based on *HyPi* stages and compare the performance of the two implementations. There are three reasons for choosing this application: 1) The pipeline pattern is not obviously applicable to the parallelization problem at hand, and represents a class of problems where an efficient pipeline implementation is not straightforward; 2) The nature

```
Input  : Matrices: X, Y – Vectors: indX, indY
         Integer: λ
Output: Ordered pairs of column indices:
         {(x, y) | x ∈ indX and y ∈ indY}
foreach colX: column in X do
    foreach colY: column in Y do
        candidate = colX ∨ colY;
        numNonZeros = popcount(candidate);
        if numNonZeros ≤ λ then
            store pair (indX[colX], indB[colY]);
        end
    end
end
```

Fig. 2: Combinatorial candidate generation algorithm. $indX$, $indY$ contain column indices of **X** and **Y** respectively; $\vee$ is the bitwise OR operation; λ is a threshold value (as described in Section 3.1).

of this algorithm is different from those for which pipeline implementations are typically used (such as those mentioned in Section 1.1), and therefore presents additional challenges; and 3) We can compare *HyPi* performance against a custom-tailored and optimized pipeline for a complex algorithm.
The hybrid pipeline implementation of combinatorial candidate generation (as described in [14]) divides the algorithm into two phases: 1) *Generate*; and 2) *Map*. The *Generate* phase is implemented as a CUDA *Device* kernel. It is responsible for generating all candidates, computing the *popcount* values, and processing the condition to verify if the result vector should be kept for elementariy testing. Results from the *Generate* phase are stored in a bit array. The *Map* phase takes this bit array and maps the results to the corresponding column indices in **X** and **Y**. The *Map* phase is implemented as a post-processing step computed on the *Host*.

3.2 HyPi Implementation of the Candidate Generation Pipeline

The *HyPi* implementation of the combinatorial candidate generation algorithm is done using *HyPi* stages. Just the three pre-defined *HyPi* stages are used, i.e., *DeviceFilter*, *CallbackFilter* and *PostProcessFilter*. First, the kernel signature is registered with the framework, so that the *DeviceFilter* can automatically call the kernel. Then the *Map* phase is registered with the framework as a post-processing function. In this case, we chose to implement the *Map* phase as a multithreaded function parallelized using OpenMP. This is to show that even though *HyPi* uses Intel TBB for pipeline parallelism, it does not imply that the entire program must be dependent on Intel tools only.
The custom-tailored pipeline from the previous implementation [14] was encapsulated in what we call the *Maximum Resource Utilization Framework (MRU)*. *MRU* is designed

to utilize all available *Host* threads when running a pipeline. This is done by having (in addition to the pipeline implementation) a *Host*-only multithreaded version of the given algorithm. We have an OpenMP parallel version of the combinatorial candidate generation algorithm. *MRU* divides the input into two parts. One part is processed by the *Host*-only parallel implementation, and the other part is processed by the pipeline implementation. This way, no *Host* threads are idle while the algorithm is executed. The *HyPi* implementation of the candidate generation algorithm also utilizes *MRU* as a harness.

4. Evaluation

We compare *HyPi* performance against: 1) a serial implementation of the candidate generation algorithm as available in *ElMo-Comp* [22], 2) an OpenMP based *Host*-only parallel implementation, and 3) the custom-tailored pipeline executed inside *MRU*. All our implementations are based on the *ElMo-Comp* code base.

4.1 Test Environment

The machine used for performance comparison consists of an Intel Nehalem EX architecture based quad-core Xeon E5520 CPU with 4 cores, where each core supports 2 hardware threads. The machine is equipped with 17GB of RAM. In addition to the CPU, the machine has an NVIDIA GTX 680 GPU with 4GB of device memory, supporting compute capability 3.0. All tests were carried out with CUDA driver version 5.5. However, it is possible to use the framework with CUDA 5.0, which is the earliest version supporting the *callback* functionality. The operating system used for the experiments is Ubuntu 12.04 LTS. The code was compiled using GCC 4.6.3 and NVCC 5.5.

4.2 Results

Given the machine available (as described in Section 4.1), it was not possible to conduct performance comparisons using real biological networks. This is because even for smaller networks, memory requirements are too high, and only three or four networks can be evaluated. This results in a very small sample against which performance comparisons can be done. In order to have a larger number of network samples, datasets of controlled sizes were artificially generated. Moreover, since we are concerned only with the combinatorial candidate generation part of the *Nullspace* algorithm, our measure of network size is the number of candidate vectors generated during the execution of the candidate generation algorithm.

Figure 3 presents a comparison of the serial, *Host*-only parallel and *HyPi* versions of the code. The plot indicates that as the number of candidates increases, the margin with which *HyPi* outperforms the other implementations gets wider. This behavior is expected, since it was shown in [14] that a pipeline implementation coupled with *MRU* is superior to the other implementations.

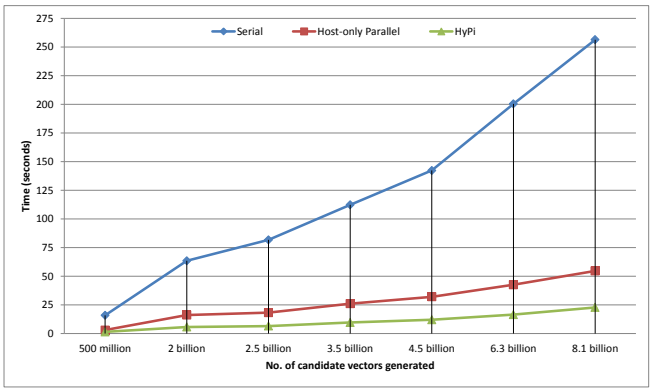

Fig. 3: Performance comparison between serial, *Host*-only parallel and *HyPi* implementations.

Figure 4 plots the performance of the custom-tailored OpenMP based pipeline implementation (designed during previous work [14]) with the *HyPi* implementation. As the plot depicts, *HyPi* performance is on par with the custom tailored pipeline. In fact, *HyPi* performance is slightly better. Although the difference is only marginal, we thought it necessary to investigate the cause.

Even though most of the pipeline design features are shared among the two implementations, management of *tokens in flight* is different. For *HyPi*, this is handled by the underlying TBB framework. In *tbb::pipeline*, the number of tokens in flight is specified by the programmer. The *HyPi* results plotted in Figure 4 are based on pipeline execution with two tokens in flight (if we change the number of tokens in flight to one, the resulting *HyPi* performance is poorer than the custom-tailored pipeline, which is expected because a single token essentially serializes the pipeline). The custom-tailored pipeline on the other hand has tightly-coupled stages without an explicit notion of tokens in flight. We believe that the performance improvement seen in *HyPi* is due to a superior token management strategy implemented in Intel TBB.

5. Conclusion and Future Work

5.1 Summary

We have presented the *Hybrid Pipeline Framework (HyPi)* intended to simplify the process of implementing pipeline parallelism in hybrid CPU-GPU architectures. The framework is implemented in C++ using Intel TBB and NVIDIA CUDA, and is suitable for pipeline applications where some stages execute on the CPU, while others execute on the GPU. *HyPi* exposes pre-developed stages as well as library routines that automate the processes of data partitioning, asynchronous data transfer from CPU-to-GPU and GPU-to-CPU, *callback* mechanism for communication between GPU and CPU, as well as automated execution of the CUDA

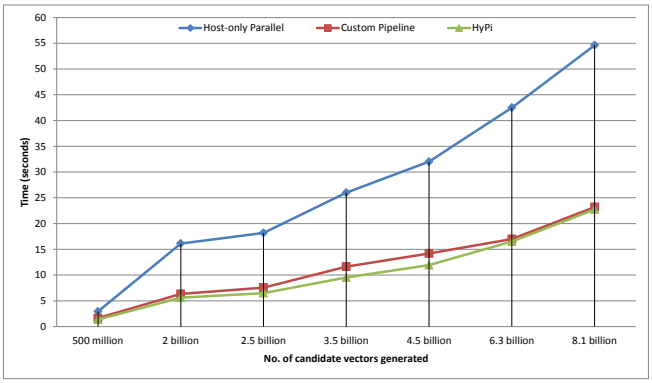

Fig. 4: Performance comparison between *Host*-only parallel, custom-tailored pipeline and *HyPi* implementations.

kernel over multiple data partitions. We have evaluated the performance of the framework against a real-world application from computational biology, and shown that it performs on par with a custom-tailored pipeline for the same problem.

5.2 Discussion

In the future, we intend to combine *HyPi* and *MRU* (mentioned in Section 3.2). At the moment, static load balancing provided by *MRU* cannot be used within the pipeline, i.e., an individual pipeline stage cannot be replicated on both the CPU and GPU for load balancing. Instead, the entire pipeline has to be replicated as CPU code and executed in parallel using OpenMP threads. Moreover, we believe it is important to introduce *dynamic load balancing*, since not all workloads can be dealt with using static schemes.

Similarly, the automated partitioning algorithms will be extended to support more complex data structures. At the moment only linear data structures are supported. Once these features have been implemented, we intend to test the framework with other scientific simulations that present different challenges in terms of hybrid pipelining.

In Section 1, we introduced the term *productivity vs. performance trade-off*. This term encapsulates our long-term research objectives. Our hypothesis is that various characteristics of a computational kernel such as *Degree of Parallelism*, *Arithmetic Intensity*, *Degree of Control Divergence* etc., make the kernel suitable for execution on either the CPU or an accelerator such as the GPU. We conjecture that a scientific simulation can be broken down into multiple computational kernels, where each kernel constitutes a stage in a hybrid pipeline. The assignment of stages to different execution architectures will depend on the above mentioned and certain other characteristics. Therefore a scientific simulation could be executed as a hybrid pipeline. Work is in progress to investigate this hypothesis, and results will be presented in a later publication.

References

[1] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, "CULA: hybrid GPU accelerated linear algebra routines," pp. 770 502–770 502–7, 2010.

[2] NVIDIA, "CUBLAS Library," User Guide DU-06702-001_v5.5, July 2013.

[3] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[4] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.

[5] L. Chen, X. Huo, and G. Agrawal, "Accelerating MapReduce on a coupled CPU-GPU architecture," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 25.

[6] M. T. Garba and H. González-Vélez, "Asymptotic peak utilisation in heterogeneous parallel CPU/GPU pipelines: A decentralised queue monitoring strategy," *Parallel Processing Letters*, vol. 22, no. 02, p. 1240008, 2012.

[7] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software pipelining," *ACM Comput. Surv.*, vol. 27, no. 3, pp. 367–432, Sept. 1995.

[8] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, Eds., *Sourcebook of Parallel Computing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[9] Y. Chen, Z. Qiao, S. Davis, H. Jiang, and K.-C. Li, "Pipelined multi-GPU MapReduce for Big-Data processing," in *Computer and Information Science*, ser. Studies in Computational Intelligence, R. Lee, Ed. Springer International Publishing, 2013, vol. 493, pp. 231–246.

[10] J. Stuart and J. Owens, "Multi-GPU MapReduce on GPU clusters," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 1068–1079.

[11] M. Xie, K.-D. Kang, and C. Basaran, "Moim: A multi-GPU MapReduce framework."

[12] NVIDIA, "CUDA C programming guide," Design Guide PG-02829-001_v6.0, February 2014.

[13] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing," *Parallel Processing Letters*, vol. 21, no. 02, pp. 245–272, 2011.

[14] F. Khalid, Z. Nikoloski, P. Tröger, and A. Polze, "Heterogeneous combinatorial candidate generation," in *Euro-Par 2013 Parallel Processing*, ser. Lecture Notes in Computer Science, F. Wolf, B. Mohr, and D. Mey, Eds. Springer Berlin Heidelberg, 2013, vol. 8097, pp. 751–762.

[15] M. Goli, M. Garba, and H. González-Vélez, "Streaming dynamic coarse-grained CPU/GPU workloads with heterogeneous pipelines in FastFlow," in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, June 2012, pp. 445–452.

[16] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[17] K. O. W. Group, "The OpenCL specification, Standard Specification," December 2011.

[18] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[19] C. Wagner, "Nullspace approach to determine the elementary modes of chemical reaction systems," *The Journal of Physical Chemistry B*, vol. 108, no. 7, pp. 2425–2431, 2004.

[20] D. Jevremović, C. T. Trinh, F. Srienc, and D. Boley, "On algebraic properties of extreme pathways in metabolic networks," *Journal of Computational Biology*, vol. 17, no. 2, pp. 107–119, 2010.

[21] J. Gagneur and S. Klamt, "Computation of elementary modes: a unifying framework and the new binary approach," *BMC bioinformatics*, vol. 5, no. 1, p. 175, 2004.

[22] D. Jevremović, C. T. Trinh, F. Srienc, C. P. Sosa, and D. Boley, "Parallelization of nullspace algorithm for the computation of metabolic pathways," *Parallel Computing*, vol. 37, no. 6-7, pp. 261–278, 2011.