In Proceedings of the International Symposium on Rapid System Prototyping, Karlsruhe, Germany, May, 2011.

Applying Graphics Processor Acceleration in a Software Defined Radio Prototyping Environment

William Plishker, George F. Zaki, Shuvra S. Bhattacharyya

Dept. of Electrical and Computer Engineering
and Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland
{plishker,gzaki,ssb}@umd.edu

Charles Clancy, John Kuykendall

Laboratory for Telecommunications Sciences
College Park, Maryland, USA
{clancy, jbk}@ltsnet.net

***Abstract*—With higher bandwidth requirements and more complex protocols, software defined radio (SDR) has ever growing computational demands. SDR applications have different levels of parallelism that can be exploited on multicore platforms, but design and programming difficulties have inhibited the adoption of specialized multicore platforms like graphics processors (GPUs). In this work we propose a new design flow that augments a popular existing SDR development environment (GNU Radio), with a dataflow foundation and a stand-alone GPU accelerated library. The approach gives an SDR developer the ability to prototype a GPU accelerated application and explore its design space fast and effectively. We demonstrate this design flow on a standard SDR benchmark and show that deciding how to utilize a GPU can be non-trivial for even relatively simple applications.**

I. Introduction

GNU Radio [1] is a software development framework that provides software defined radio (SDR) developers a rich library and a customized runtime engine to design and test radio applications. GNU Radio is extensive enough to describe audio radio transceivers, distributed sensor networks, and radar systems, and fast enough to run such systems on off-the-self radio hardware and general purpose processors (GPPs). Such features have made GNU Radio an excellent rapid prototyping system, allowing designers to come to an initial functional implementation quickly and reliably.

GNU Radio was developed with general purpose programmable systems in mind. Often initial SDR prototypes were fast enough to be deployed on general purpose processors or needed few custom accelerators. As new generations of processors were backwards compatible with software, GNU Radio implementations could track with Moore's Law. As a result, programmable solutions have been competitive with custom hardware solutions that required longer design time and greater expense to port to the latest process generation. But with the decline in frequency improvements of GPPs, SDR solutions are increasingly in need of multicore acceleration, such as that provided by graphics processors (GPUs). SDR is well positioned to make use of them since many SDR applications have abundant parallelism.

GPUs are starting to be employed in SDR solutions, but their adoption has been inhibited by a number of difficulties, including architectural complexity, new programming languages, and stylized parallelism. While other research is addressing these topics [5] [6], one of the primary barriers in many domains is the ability to quickly prototype the performance advantages of a GPU for a particular application. The inability to assess the performance impact of a GPU with an initial prototype leaves developers to doubt if the time and expense of targeting a GPU is worth the potential benefit.

Many design decisions are needed before arriving at initial multicore prototype including mapping tasks to processors and data to distributed memories. Mapping SDR applications is further complicated by application requirements. The amount of parallelism present may be dictated by the application itself based on its latency tolerances and available vectorization of the kernels. More vectorization tends to lead to higher utilization of the platform (and therefore higher throughput), but often at the expense of increased latency and buffer memory requirements. Also an accelerator typically requires significant latency to move data to or from the host processor, so sufficient data must be burst to the accelerator to amortize such overheads.

Ideally, application designers would be simply presented with a Pareto curve of latency versus vectorization trade-offs so that an appropriate design point can be selected. However, vectorization generally influences the efficiency of a given mapping. Thus, to fully unlock the potential of heterogeneous multiprocessor platforms for SDR, designers must be able to arrive at a variety of solutions quickly, so that the design space may be explored along such critical dimensions.

To enable developers to arrive at an initial prototype that utilizes GPUs, we introduce a new SDR design flow, as shown in Figure 1. We begin with a formal description of an SDR application, which we extract from a GNU Radio specification. Formalisms provide the design flow with a structured, portable application description which can be used for vectorization,

978-1-4577-0660-8/11/$26.00 © 2011 IEEE

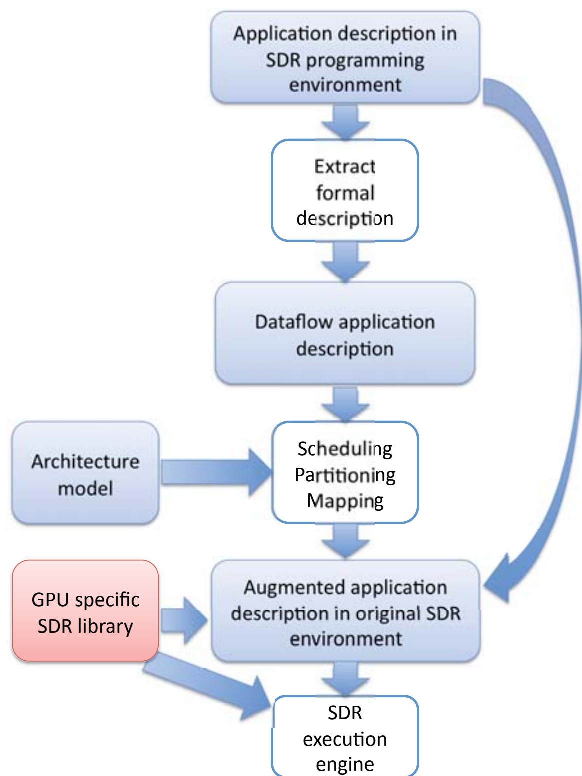

Fig. 1. Dataflow founded SDR Design Flow.

latency, and other design decisions. These design decisions can ultimately be incorporated into an SDR application through a GPU specific library of SDR actors. For this work, we have constructed GRGPU, which is a such a library written for GNU Radio. With this design process, we demonstrate the value of this approach with GNU Radio benchmark on a platform with a GPU.

II. BACKGROUND

Dataflow graphs are widely used in the modeling of signal processing applications. A dataflow graph G consists of set of vertexes V and a set of edges E. The vertices or *actors* represent computational functions, and edges represent FIFO buffers that can hold data values, which are encapsulated as *tokens*. Depending on the application and the required level of model-based decomposition, actors may represent simple arithmetic operations, such as multipliers or more complex operations as turbo decoders.

A directed edge $e(v_1, v_2)$ in a dataflow graph is an ordered pair of a source actor $v_1 = src(e)$ and sink actor $v_2 = snk(e)$, where $v_1 \in V$ and $v_2 \in V$. When a vertex v executes or *fires*, it consumes zero or more tokens from each input edge and produces zero or more tokens on each output edge. Synchronous Data Flow (SDF) [8] is a specialized form of dataflow where for every edge $e \in E$, a fixed number of tokens is produced onto e every time $src(e)$ is invoked, and similarly, a fixed number of tokens is consumed from e every time $snk(e)$ is invoked. These fixed numbers are represented, respectively, by $prd(e)$ and $cns(e)$. Homogeneous Synchronous Data Flow (HSDF) is a restricted form of SDF where $prd(e) = cns(e) = 1$ for every edge e.

Given an SDF graph G, a *schedule* for the graph is a sequence of actor invocations. A *valid schedule* guarantees that every actor is fired at least once, there is no deadlock due to token underflow on any edge, and there is no net change in the number of tokens on any edge in the graph (i.e., the total number of tokens produced on each edge during the schedule is equal to the total number consumed from the edge). If a valid schedule exists for G, then we say that G is *consistent*. For each actor v in a consistent SDF graph, there is a unique *repetition count* $q(v)$, which gives the number of times that v must be executed in a minimal valid schedule (i.e., a valid schedule that involves a minimum number of actor firings). In general, a consistent SDF graph can have many different valid schedules, and these schedules can differ widely in the associated trade-offs in terms of metrics such as latency, throughput, code size, and buffer memory requirements [4].

III. RELATED WORK

Many models of computation have been suggested to describe software radio systems. In [2], the advantages and drawbacks of various models are investigated. Also different dataflow models that can be applied to various actors of an LTE receiver are demonstrated.

Actor implementation on GPUs is discussed in [13]. A GPU compiler is described in order to take a naive actor implementation written in CUDA [11], and generate an efficient kernel configuration that enhances the load balance on the available GPU cores, hides memory latency, and coalesces data movement. This work can be used in our proposed framework to enhance the implementation of individual software radio actors on a GPU. Raising the abstraction of CUDA programming through program analysis is the focus of Copperhead [6].

In [12], the authors present a multicore scheduler that maps SDF graphs to a tile based architecture. The mapping process is streamlined to avoid the derivation of equivalent HSDF graphs, which can involve significant time and space overhead. In more general work, MpAssign [5] employs several heuristics, allows different cost functions and architectural constraints to arrive at a solution.

In [15], a dynamic multiprocessor scheduler for SDR applications is described. The basic platform consists of a Universal Software Radio Peripheral (USRP), and cluster of GPPs. A flexible framework for dynamic mapping of SDR components onto heterogeneous multiprocessor platforms is described in [9].

Various heuristics and mixed linear programming models have been suggested for scheduling task graphs on homogeneous and heterogeneous processors (e.g., see [10]). In these works, the problem formulations are developed to address different objective functions and target platforms for implementing the input application graphs.

The focus of this work is to construct a backend capable of integrating specialized multicore solutions into a domain

specific prototyping environment. This should facilitate the previously described dataflow based design flow, but should also enable these other works to be applied in the field of SDR. Any solution targeting a complex multicore system is unlikely to produce the optimal solution with its first implementation. The ability to quickly generate and evaluate many solutions on a multicore platform should improve the efficacy the approach and ultimately the quality of the final solution.

IV. SDR Design Flow for GPUs

We implemented the design flow proposed in Figure 1 by using GNU Radio as the SDR description and runtime environment and the Dataflow Interchange Format (DIF) [7] for the dataflow representation and associated tools. Our GPU target was CUDA enabled NVIDIA GPUs. With these tools in place the design flow proceeds as described in the following steps:

1) Designers write their SDR application in GNU Radio with no consideration for the underlying platform. As GNU Radio has an execution engine and a library of SDR components, designers can verify correct functionality of their application. For existing GNU Radio applications, nothing must be changed with the description to continue with the design flow.
2) If actors of interest are not in the GPU accelerated library, a designer writes accelerated versions of the actors in CUDA. The design focuses on exposing the parallelism to match the GPU architecture in as parametrized way as possible.
3) Either through automated or manual processes, instantiated actors are either assigned to a GPU or designated to remain on a GPP. With complex trade offs between GPU and GPP assignments possible, this step may be revisited often as part of a system level design space exploration. Dataflow provides a platform independent foundation for analytically determining good mappings, but designer insight is also a valuable resource to be utilized at this step.
4) The mapping result is utilized by augmenting the original SDR application description environment. By leveraging a stand-alone library of CUDA accelerated actors for GNU Radio, the designer can describe and run the accelerated application description with existing design flow properties.

The following sections cover these steps in detail, specifically as they relate to our instance of the design flow that utilizes CUDA, GNU Radio, and DIF.

A. Writing GPU Accelerated Actors

After the application graph is described in GNU Radio, actors are individually accelerated using GPU specific tools. If an actor of interest is not present in the GPU accelerated library, the developer switches to the GPU customized programming environment, which in our case is CUDA. The designer is still saddled with difficult design decisions, but these decisions are localized to a single actor. System level design decisions are orthogonal to this step of the design process. While we do not aim to replace the programming approach of the actors functionality, the following design strategy lends itself to later design space exploration by the developer.

As with other GPU programming environments, in CUDA a designer must divide their application into levels of parallelism: *threads* and *blocks*, where threads represent the smallest unit of a sequential task to be run in parallel and blocks are groups of threads. In our experience, SDR actors vary in how to use thread level parallelism, but tend to realize block level parallelism with parallelism at the sample level. The ability to tightly couple execution between threads within a block creates a host of possibilities for the basic unit of work within a block, be it processing a code word, multiplying and accumulating for a tap, or performing an operation on a matrix. Because blocks are decoupled, only fully independent tasks can be parallelized. For SDR those situations tend to arise between channels or between samples on a single channel. Some samples may overlap between blocks to support the processing of a neighboring sample, but this redundancy is often more than offset by the performance benefits of parallelization.

The performance of this parallelization strategy strongly influenced by the number of channels or the size of a chunk of samples that can be processed at one time. When the application requests processing on a small chunk of sample, there are few blocks to spread across a GPU leaving it under utilized, while large chunks enable high-utilization. The performance difference between small and large chunks is non-linear due to the high fixed latency penalty that both scenarios experience when transferring data to and from the GPU and launching kernels. When chunks are small, GPU time is dominated by transfer time, but when chunks are larger, computation time of the kernel dominates, which amortizes the fixed penalty delay. As the application dictates these values, actors must be written in a parametrized way to accommodate different size inputs.

B. Partitioning, Scheduling, and Mapping

Once actors are written, system level design decisions must be made, such as assigning which actors are to invoke GPU acceleration. With some applications, the best solution may be to offload every actor that is faster on the GPU than it is on the GPP. But in some cases, this greedy strategy fails to recognize the work that could occur simultaneously on the GPP, while the host thread with the kernel call waits for the GPU kernel to finish. A general solution to the problem would consider application features such as rates of firings, dependencies, and execution times on each platform of each actor, as well as architectural features such as the number and types of processing elements, memories, and topology. To simplify the problem, designers can cluster certain actors together so that they are assigned to the same. To promote this clustering, designers may partition the application graph.

Multirate applications also need to be scheduled properly to ensure firing rates and dependencies are proper accounted for.

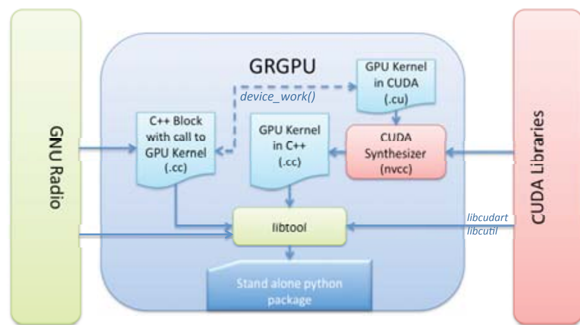

Fig. 2. GRGPU: A GNU Radio integration of GPU accelerated actors.

When the application can be extracted into a formal dataflow model, schedulers will not only respect these constraints but are able to optimize for buffer assignments [3]. The applicability of such techniques for specialized multicore platforms are still open research, and this design flow enables greater experimentation with them for SDR applications. Manual scheduling and mapping is likely to continue to dominate smaller, more homogeneous mappings, but a grounding in dataflow opens the door for new automation techniques. In this work we focus on the design flow, conventions for writing SDR actors, and integrating GPU accelerated actors with GNU Radio.

C. GRGPU: GPU Acceleration in GNU Radio

We developed GPU accelerated GNU Radio actors in a separate, stand-alone library called *GRGPU*. GRGPU extends GNU Radio's build and install framework to link against libraries in CUDA as shown in Figure 2. After building against CUDA libraries, the resulting actors may be instantiated alongside traditional GNU Radio actors, meaning that designers may swap out existing actors for GRGPU actors to bring GPU acceleration to existing SDR applications. The traditional GNU Radio actors run unaffected on the host GPP, while GRGPU actors utilize the GPU.

When writing a new GRGPU actor, application developers start by writing a normal GNU Radio actor including a C++ wrapper that describes the interface to the actor. The GPU kernels are written in CUDA in a separate file and tied back to the C++ wrapper via C functions such as *device_work()*. Additional configuration information may be sent in through the same mechanism. For example, the taps of a FIR filter typically need to be updated only once or rarely during the execution, so instead of passing the tap coefficients during each firing of the actor (taps sent from *work()* to *device_work()* to the kernel call), they could be loaded into device memory when the taps are updated in GNU Radio. The CUDA compiler, NVCC, is invoked to synthesize C++ code which contains binaries of the code destined for the GPU, but glue code formatted for C++. By generating the C++ instead of an object file directly, we are able to make use of the standard GNU build process using libtool. Even though the original application description was in a different language, the code is wrapped and built in the GNU standard way giving it compatibility with previous and future versions of GNU and GNU Radio.

When a GNU Radio actor is instantiated, a new C++ object is created which stores and manages the state of the actor. However, state in the CUDA file is not automatically replicated, creating a conflict when more than one GRGPU actor of the same type is instantiated. To work around this issue, we save CUDA (both host and GPU) state inside the C++ actor, which includes GPU memory pointers of data already loaded to the GPU. The state from the GPU itself is not saved inside the C++ object, but rather the pointers to the device memory are. Data residing in the GPUs memory space is explicitly managed on the host, so saving GPU pointers is sufficient for keeping the state of the CUDA portion of an actor.

To minimize the number of host-to-GPU and GPU-to-host transfers, we introduce two actors, *H2D* and *D2H*, to explicitly move data to and from the device in the flow graph. This allows other GRGPU actors to contain only kernels that produce and consume data in the GPU memory. If multiple GPU operations are chained together, data is processed locally, reducing redundant I/O between GPU and host as shown in Figure 3. In GNU Radio, the host side buffers still exist which connect links between the C++ objects that wrap the CUDA kernels. Instead of carrying data, these buffers now carry pointers to data in GPU memory. From a host perspective, *H2D* and *D2H* transform host data to and from GPU pointers, respectively.

While having both a host buffer and a GPU buffer introduces some redundancy, it has a number of benefits which make this an attractive solution. First, there is no change to the GNU Radio engine. The GNU Radio engine still manages data being produced and consumed by each actor, so decisions on chunk size or invocation order do not need to be changed with the use of GRGPU actors. Second, GPU buffers may be safely managed by the GRGPU actors. With GPU pointers being sent through host buffers, actors need only concern themselves with maintaining their own input and output buffers. This provides dynamic flexibility (actors can choose to create and free memory for data as needed) or static performance tuning (actors can maintain circular buffers which they read and write a fixed amount of data to and from). Such schemes require coordination between GRGPU actors and potentially information regarding buffer sizing, but the designer does have the power to manage these performance critical actions without redesigning or changing GRGPU. Future versions of GRGPU could provide a designers with a few options regarding these schemes and even make use of the dataflow schedule or other analysis to make quality design decisions. Finally, no extraneous transfers between GPU and host occur. While the host and GPU buffers mirror each other, no transfers occur between them, which avoids I/O latencies that can be the cause of application bottlenecks.

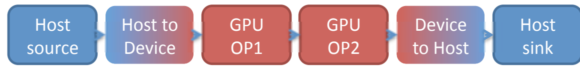

Fig. 3. GRGPU actors within H2D and D2H communicate data using the GPUs memory, avoiding unnecessary host/GPU transfers

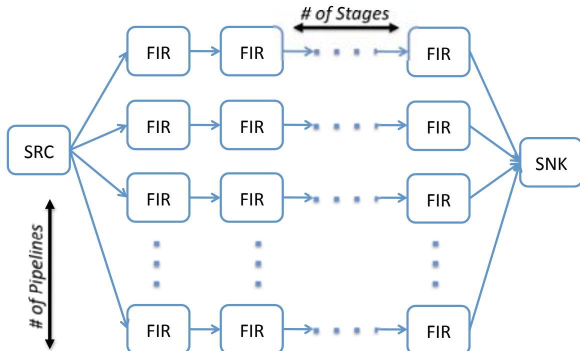

Fig. 4. SDF graph of the mp-sched Benchmark.

V. Evaluation

We have experimented with the proposed design flow using the mp-sched benchmark. Figure 4 shows the mp-sched benchmark pictorially. Each of the actors after the distributor performs FIR filtering. To provide flexibility for evaluating different multicore platforms, it is configurable with number of chains of FIR filters (*pipelines*) and the depth of the chains (*stages*). This benchmark describes a flow graph that consists of a rectangular grid of FIR filters. The dimensions of this grid are parametrized by the *number of stages* ($STAGES$) and *number of pipelines* ($PIPES$). The total number of FIR filters is thus equal to $PIPES \times STAGES$. This benchmark represents a non-trivial problem for the multiprocessor scheduler as all actors in different pipelines can be executed in parallel. More information about the mp-sched benchmark can be found in [1].

A. FIR Filter Design

In this implementation [14], we take advantage of data parallelism between the filter output samples as well as functional parallelism to calculate every sample. For relatively large chunks of samples, the CUDA kernel is configured such that the number of blocks is equal to double the number of available streaming multiprocessors. By using this configuration, the first level of data parallelism can be achieved if every CUDA block is responsible to calculate a different set of output samples. In other words, the required number of output samples are evenly distributed on the number of CUDA blocks. To overcome the inherited stateful property of the FIR filter(i.e, consecutive output samples depend on some shared input samples), the input of every block must contain an extra set of delayed input samples equal to the number of taps.

To reduce the number of device memory access, initially all of the threads will perform a load of a coalesced chunk of input elements to the shared memory of a multiprocessor. Then every thread will be responsible of calculating the product of a filter tap coefficient with an input sample, and adding this product to the partial sum of the previous stage. After processing a set of inputs, the threads perform a block store of the calculated results to the GPU device memory.

B. Empirical Results

We found a variety of design points of mp-sched to evaluate the utility of rapid prototyping with GRGPU. The target platform was two GPPs (Intel Xeon CPUs 3GHz) and a GPU (an NVidia GTX 260). The actors performed a 60 tap FIR filtering with either CUDA acceleration in the case of GPU accelerated actors or SSE acceleration in the case of the GPP. To minimize the latencies incurred by using *H2D* and *D2H*, the GPU accelerated actors were clustered together leaving remaining GPP actors similarly clustered.

In the case of our exploration of the mp-sched implementation design space, each pipeline was located in a separate thread and the number of actors with GPU acceleration was configurable. Mp-sched pipelines could run in parallel and share the GPU as an acceleration resource during runtime. Multiple pipelines with GPU accelerated actors were forced to serialize their GPU accesses according to CUDA conventions. For example, one possible solution to a 2x20 instance of mp-sched is shown in Figure 5. The Gantt chart is not to scale, but shows how the two different pipelines (one in red and one in blue), are able to run in parallel on the two GPPs, but must have exclusive access to the GPU when running accelerated actors. While the cross thread sequencing was not specified at runtime, GRGPU's ability to specify acceleration and clustering enables the creation of multicore, GPU accelerated complete solutions.

The problem for a designer is then to leverage GPPs and GPU, weigh SSE acceleration and CUDA acceleration, account for communication latencies between GPU and GPP and thread to thread, and consider how all of this will occur in parallel. Models and automated techniques should continue to assist in providing good starting points, but a necessary condition to arriving at a quality solution is still the ability to try many points quickly.

To this end, we constructed an illustrative example that produces an interesting set of design points: mp-sched with 20 stages and varied the number of pipelines. Figure 6 shows a sub-sampling of the design space. "All GPP" means all stages of all pipelines are assigned to the GPPs, while "All GPU" means all stages of all pipelines are assigned to the GPU. "3/4 GPP", "Half GPP", and "3/4 GPU" indicates that three quarters, one half, or one quarter of the stages of all pipelines are assigned to the GPP, respectively, while the remaining actors use the GPU. For example, Figure 5 shows the 2x20 Half GPP solution. We also evaluated solutions in which one of the pipelines was all GPP and the rest GPU ("One GPP") and the reverse ("One GPU"). In the case of only one pipeline, these solutions were equivalent to an all GPP or all GPU solution. We ran each solution for 200,000 samples and recorded the execution time, including GNU Radio overheads, communication overheads, etc.

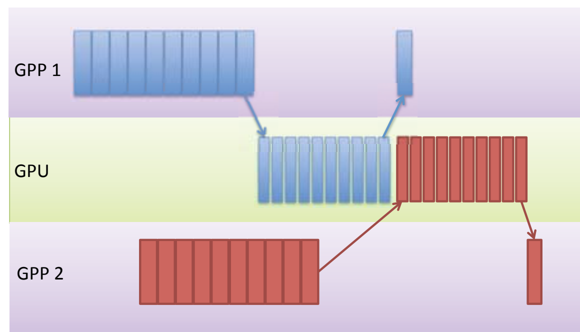

Fig. 5. Gantt chart for 2x20 mp-sched graph on 2 GPP and 1 GPU. The blue and the red set of blocks and arrows each represent one branch of the mp-sched instance.

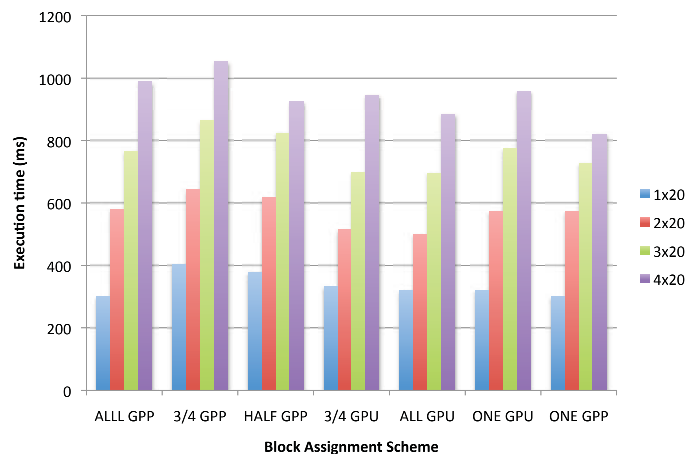

Fig. 6. A sampling of the design space of 1x20, 2x20, 3x20, 4x20 mp-sched graph on 2 GPPs and 1 GPU for different assignments.

For the 60 tap FIR filter, SSE acceleration performs well, but still somewhat slower than the GPU implementation, so once a sufficient amount of computation is located on the GPU, GPU weighted implementations tend to perform better. But this graph does reveal that the GPU should be employed in different ways depending on the number of pipelines. For example, a single pipeline implies that there is not quite enough computation present to merit GPU acceleration. However when 2 or more pipelines are used, the GPPs become saturated to the point that GPU acceleration can improve upon the result. When 4 pipelines are needed, one GPP only pipeline proves higher performing than an all GPU solution, indicating that the GPU itself has become saturated with computation and that employing more of the GPP is appropriate. In each of the cases, retrospective reasoning gives us insight into improving performance, but a change in GPU, communication latencies, etc. would likely change this space again, leaving a designer to re-explore the design space.

It should be possible to arrive at these solutions more analytically to accelerate the design space exploration, but inevitably a set of points will need to be evaluated to judge the efficacy of any analytical assistance. GRGPU will continue to provide value in such a scenario feeding-back empirical solutions to the design space exploration engine.

VI. Conclusion and Future Work

As SDR attempts to leverage more special purpose multi-core platforms in complex applications, application developers must be able to quickly arrive at an initial prototype to understand the potential performance benefits. In this paper, we have presented a design flow that extends a popular SDR environment, lays the foundation for rigorous analysis from formal models, and creates a stand-alone library of GPU accelerated actors which can be placed inside of existing applications. GPU integration into an SDR specific programming environment allows application designers to quickly evaluate GPU accelerated implementations and explore the design space of possible solutions at a system level.

Useful directions for future work include new methods for dealing with scheduling, partitioning, and mapping for multicore systems along with evaluating existing automation solutions that have been developed. Also, GRGPU should be able to extend to multi-GPU platforms by customizing GRGPU actors to communicate and launch on a specific GPU.

Acknowledgments

This research was sponsored in part by the Laboratory for Telecommunication Sciences, and Texas Instruments.

References

[1] http://gnuradio.org/redmine/wiki/gnuradio. Nov 2010.
[2] H. Berg, C. Brunelli, and U. Lucking. Analyzing models of computation for software defined radio applications. In *Proc. IEEE International Symposium on System-on-Chip*, pages 1–4, Nov. 2008.
[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.
[4] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 21(2):151–166, June 1999.
[5] Y. Bouchebaba, P. Paulin, A. E. Ozcan, B. Lavigueur, M. Langevin, O. Benny, and G. Nicolescu. Mpassign: A framework for solving the many-core platform mapping problem. In *Rapid System Prototyping (RSP)*, June 2010.
[6] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. Technical Report UCB/EECS-2010-124, EECS Department, University of California, Berkeley, Sep 2010.
[7] C. Hsu, I. Corretjer, M. Ko., W. Plishker, and S. S. Bhattacharyya. Dataflow interchange format: Language reference for DIF language version 1.0, user's guide for DIF package version 1.0. Technical Report UMIACS-TR-2007-32, Institute for Advanced Computer Studies, University of Maryland at College Park, June 2007. Also Computer Science Technical Report CS-TR-4871.
[8] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.
[9] V. Marojevic, X. R. Balleste, and A. Gelonch. A computing resource management framework for software-defined radios. *IEEE Transactions on Computers*, 57:1399–1412, 2008.
[10] R. Niemann and P. Marwedel. Hardware/software partitioning using integer programming. In *Proc. of the European Design and Test Conference*, pages 473 –479, Mar. 1996.
[11] NVIDIA. CUDA C programming guide version 3.1.1. July 2010.
[12] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proc. of the 44th annual Design Automation Conference, DAC '07*, pages 777–782, June 2007.

[13] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proc. of the 2010 ACM SIGPLAN conference on Programming language desing and implementation*, June 2010.

[14] G. Zaki, W. Plishker, T. OShea, N. McCarthy, C. Clancy, E. Blossom, and S. S. Bhattacharyya. Integration of dataflow optimization techniques into a software radio design framework. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 243–247, Pacific Grove, California, November 2009. Invited paper.

[15] K. Zheng, G. Li, and L. Huang. A weighted-selective scheduling scheme in an open software radio environment. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 561 –564, Aug 2007.