A Heterogeneous GASNet Implementation for FPGA-accelerated Computing

Ruediger Willenberg
Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada
Email: willenbe@ece.toronto.edu

Paul Chow
Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada
Email: pc@ece.toronto.edu

Abstract—**This paper introduces an effort to incorporate reconfigurable logic (FPGA) components into the Partitioned Global Address Space model. For this purpose, we have implemented a heterogeneous implementation of GASNet that supports distributed applications with software and hardware components and easy migration of kernels from software to hardware. We present a use case and preliminary performance numbers.**

I. Motivation

Field-Programmable Gate Arrays (FPGAs) are a valuable tool to accelerate High-Performance Computing (HPC) systems. Since virtually any digital circuit can be implemented on an FPGA, it is possible to freely design pipelines and dataflows tailored to any computation problem. This degree of freedom is balanced with comparably low maximum clock speeds in contrast to CPUs and GPUs. Nevertheless, the available design flexibility means that FPGAs can outcompete CPUs and GPUs on a number of computations. Furthermore, they can do so using less energy per computation. The possibility to completely reconfigure an FPGA in milliseconds to a few seconds means that they can be productively used in common time- and resource-shared computing clusters.

Despite these advantages, very few HPC systems employ FPGAs. The major reason for this is that, from the perspective of a software programmer, they appear to be much harder to program than CPUs and GPUs. The first problem is that applying the classical Von-Neumann programming model to FPGAs does not make sense because doing so would constrain their major asset, the ability to combine logic and memory into arbitrary, customized dataflows.

Secondly, these dataflows have to be described in very low-level Hardware Description Languages (HDLs) like VHDL and Verilog that are somewhat mismatched to the synthesis problem for historical reasons; while several improved HDL idioms have been introduced recently, FPGA and tool vendors have proven to be very conservative in introducing support for them.

Third and last, there is no unified standard to interface with a host of different FPGA devices and platforms. In contrast, GPUs have been made accessible to a large user community by the OpenCL[1] and CUDA[2] APIs.

There is evidence of progress on the first and second problems with the introduction of High-Level Synthesis (HLS) tools, which allow the conversion of C or C++ code into digital circuits; the success of these efforts is highly dependent on the kind of code to be translated.

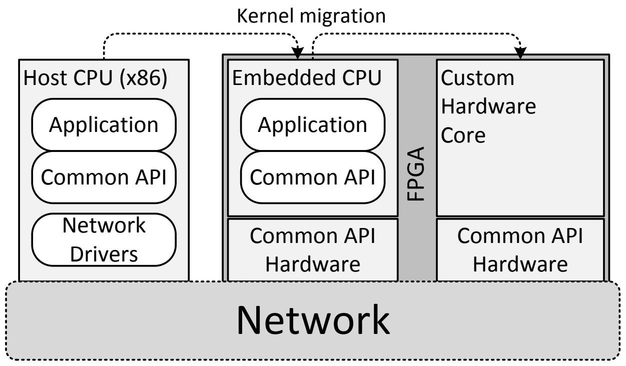

Fig. 1. Example for a multiple-platform system with unified parallel API: Host CPU, embedded CPU on FPGA and custom FPGA hardware

The work introduced in this paper focuses on the third problem, the productive integration of FPGA components into HPC systems through a standard interface. Our paper is structured as follows: Section II explains our basic philosophy for implementing CPU/FPGA heterogeneity. Section III briefly discusses related work. Our hardware and software components are introduced in Sections IV and V. Section VI demonstrates our design flow on an application example. We conclude in Section VII and present an outlook to upcoming work in Section VIII.

II. Unified Programming Model

In our opinion, a unified programming model and API for all components in a heterogeneous system, as shown in Figure 1, is beneficial to keeping applications both maintainable and scalable. In this approach, the same API is used by both application processes running on host CPUs as well as on embedded CPUs located on the FPGA. The latter can be either *soft processors* implemented in the logic fabric or hardwired cores (like the ARM processors integrated into some recent FPGA families). Furthermore, custom hardware components will use a similar “API” by using the same set of configuration parameters to control synchronization and data communication. Instead of FPGA components being utilized in the co-processor model, hardware and software components are treated as equal interacting processes.

A significant benefit of a common API is the easy migration of performance-critical application kernels to hardware. An algorithm can be implemented, verified and profiled completely in software, taking advantage of the sophisticated development and debugging tools available in this domain. When profiling has identified the code sections taking up the most execution time, these can be converted into custom hardware cores and seamlessly integrated into the original software component architecture.

Previous work has successfully demonstrated this approach using the Message Passing Interface [3]. Our current work implements the same approach for the Partitioned Global Address Space model. Besides its established productivity and scalability benefits, PGAS is a good model for the disparate memory architectures found in a heterogeneous system.

Based on its straightforward and efficient concept of *Active Messages*, we have picked GASNet [4] as the specific API to adapt to reconfigurable hardware. In the following sections, we present *Toronto Heterogeneous GASNet* (THeGASNet), our own implementation of the GASNet Core API. We have previously introduced an early version of our *GAScore* remote memory access core [5] that supports intra- and inter-FPGA communication between on-chip-memories. This paper presents the following new contributions:

- An updated *GAScore* version that supports burst access to external memory through a standard embedded bus (AXI)
- Support for *Strided* and *Vectored* Active Message types to support complex remote memory transfers
- A complete software/hardware GASNet framework that supports portability and interoperability of applications using the *THeGASNet* Core API on x86-64, ARMv7 and Xilinx MicroBlaze processors.

III. Related work

The approach of having a similar, migration-conducive API for software and hardware components has been successfully used by Saldana et al.[3]. TMD-MPI is a library that implements a subset of the MPI standard to enable message passing between FPGA and CPU components. It has successfully demonstrated a common communication model for the simulation of molecular dynamics. Our work is inspired by and still partially based on TMD-MPI infrastructure. TMD-MPI's downsides are the ones incumbent to any message-passing system: The need for low-level transfer management, two-sided communication and its scalability limits, the inefficiencies of indirect communication to remote memories and, finally, the impediments to dynamic memory accesses and the use of linked data structures.

Hthreads [6] is a hybrid shared-memory programming model that focuses on implementing FPGA hardware in the form of real-time operating system threads, with most of the properties of software, but adding real concurrency. Hthreads focuses on components sharing buses with each other in single-chip processor systems, and lacks a scalable programming model for multi-node systems.

On the PGAS side, SHMEM+ [7] is an extended version of the established SHMEM library that uses the concept of *multilevel PGAS* as defined by its authors: Every processing node has multiple levels of main memory, e.g. CPU main memory as well as an FPGA accelerator's main memory, which are accessed differently by SHMEM+. For node-to-node transfers GASNet is used, for transfers to a local or remote FPGA a vendor-specific interface has to be accessed by the local CPU.

El-Ghazawi et al.[8] examine two different approaches to use FPGAs with Unified Parallel C: In the library approach, a core library of FPGA bitstreams for specific functionalities exists. Function cores can be explicitly loaded into the FPGA. An asynchronous function call transfers data for processing into the FPGA, a later completion wait transfers the processed data back into CPU-accessible memory. The second approach uses a C-to-RTL synthesis of selected portions of the UPC code: A parser identifies *upc_forall*-statements that can be well parallelized in hardware and splits their compilation off to Impulse-C[9]; corresponding data transfers to and from the FPGA are inserted into the CPU code.

Our common concern with the two PGAS solutions is that they leave the CPU(s) in charge of all communication management, while the FPGA remains in a classic, passive accelerator role. Truly efficient one-sided communication as embraced by PGAS is therefore not available hardware-to-hardware, and FPGA capabilities might be underutilized.

Finally, RAMP Blue [10] is a massive multiprocessor platform, with the intent of supporting multiprocessor research through use of soft processors on FPGAs. Like our work, it also uses MicroBlaze processors connected through a FIFO-based network scalable across multiple FPGAs and FPGA boards. It uses the NAS Parallel Benchmarks[11] based on UPC for performance testing. To enable this, Berkeley UPC and the underlying GASNet were ported to the MicroBlaze platform and the FIFO-based conduits. While the successful porting of the full GASNet library is an impressive feat, there is no intent to extend the software implementation to hardware accelerators; the MicroBlaze itself is a stand-in for future multiprocessor cores. In contrast, our work focuses on a low-footprint GASNet Core library for the MicroBlaze that can communicate across platforms; the intent is that the MicroBlaze serves as an intermediate stage of the application design process and is later replaced by hardware accelerators that can communicate in the same way.

IV. THeGASNet Hardware Components

THeGASNet enables hardware processing units on FPGAs to act and communicate like GASNet software nodes. For this purpose, it needs to provide FPGA infrastructure that

- enables networking between GASNet hardware and software nodes on different devices
- provides *Remote Direct Memory Access* (RDMA), the capability to read from and write to shared memory segments on a different node
- facilitates the transmission and reception of GASNet *Active Messages* by a computation core.

The following sections describe the provided components in more detail.

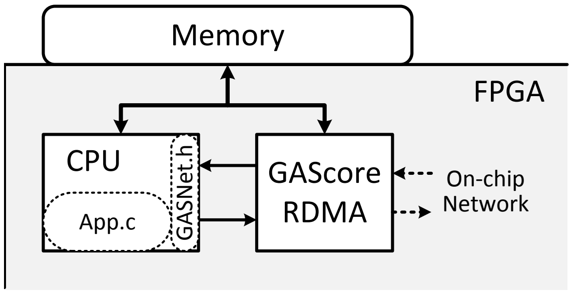

Fig. 2. GAScore RDMA engine with embedded processor

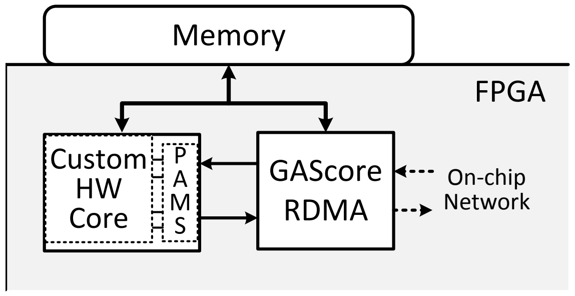

Fig. 3. GAScore RDMA engine with custom hardware processing element and active message sequencer

A. NetIf on-chip network

GASNet components use a packet on-chip network based on FIFO connections called *Fast Simplex Links* (FSLs) and routers called *NetIfs*. This network infrastructure has been established by previous work on message-passing [3].

Fast Simplex Links are a FIFO implementation that is architecture-specific to the FPGA vendor Xilinx. Each FSL has a width of at least 32 data bits and a control bit to designate header data. Bandwidth can be increased by using data widths of higher powers of two, with data width conversion enabled by straightforward packet converters. For the default depth of 16 data words, FSLs can be efficiently implemented in basic logic blocks with very low resource use. There are asynchronous FSL variants that can use different clock domains on different ends at the cost of extra latency. By employing these asynchronous FIFOs, each single component can be tuned to run at its optimum speed.

A packet consists of a single-word header specifying the source and destination GASNet nodes and the packet size, followed by the payload data. Each hardware block comprising a GASNet node or an off-chip connection is connected to its own NetIf, which connects to all other NetIfs on the chip. Because of the low resource footprint of FSLs, systems with up to ten NetIf components can be fully connected, allowing for straightforward, low-latency routing. Routing tables can be re-programmed into an FPGA bitstream without re-running the time-intensive design implementation process.

Off-chip connections to other FPGAs or to host PCs encapsulate the NetIf packets in their proprietary formats, so that transparent GASNet node-to-node communication is possible through multiple devices and PCs.

B. GAScore remote DMA engine

The central component of on-chip GASNet support is a remote DMA engine called *GAScore* (Global Address Space core) 2. GAScore manages the GASNet communication of an embedded processor or a custom hardware core by receiving and transmitting Core API *Active Messages* encapsulated in NetIf packets. A GAScore instance has three ports:

- A duplex FSL connection to a NetIF. GAScore acts as a host that can receive NetIf packets addressed to its node ID and send packets to any other GASNet node.
- A master burst connection to an AXI bus. AXI is the standard parallel bus of ARM systems and recent Xilinx FPGA embedded systems. GAScore usually shares an AXI connection to a memory with the embedded processor or custom processing core.
- Two duplex FSL connections to the processor or custom core. The processor or custom core can request GAScore to send an Active Message to another node. GAScore can notify the processor or custom core of a received Active Message and trigger the execution of a handler function.

A GAScore instance fulfills two largely concurrent functions, which are initially described here working together with an embedded processor:

1) *Transmission of Active Messages:* The processor can initiate an Active Message by sending a multi-word request including configuration parameters to the GAScore through a FIFO. The written parameters closely reflect the function arguments that a software call to a GASNet Core API function uses. As a consequence, a software GASNet implementation on an embedded processor only needs to be a thin software layer that turns function arguments into words written to the FIFO.
   If the request is for a memory-to-memory data transfer (Active Message type *Long*), GAScore will retrieve the corresponding data directly from memory, attach it to the function parameters to form a NetIf packet, and send the packet off to its destination GASNet node. GAScore will signal transmission of the data back to the processor and therefore enable the processor to modify the corresponding memory again when necessary.
2) *Reception of Active Messages:* GAScore will process incoming messages by writing any payload data to the memory location(s) specified by the Active Message arguments. It will then write out the Active Message parameters to the processor to allow software handling of the message reception. The data arriving through the FIFO will trigger an interrupt in the processor, and the transmitted arguments will be used to call the specified handler function for the message[1]. Again, the FIFO data is closely reflecting software function arguments, and therefore the software GASNet overhead is minimal.

[1]As AM function calls result in a sequence of FIFO writes that need to be monolithic, interrupts are disabled at the beginning of an AM call and re-enabled at its end. THeGASNet does not currently implement general no-interrupt sections and handler-safe locks (current applications only use GNU atomic accesses to single data words), but both could be provided without expected complications for both the single-threaded MicroBlaze and the multi-threaded x86-64 and ARM CPUs.

Beside the Active Message types *Short* (No memory payload) and *Long* (with memory payload), GAScore also supports:

- *Medium* messages: In the software domain, these messages are used to transport data to a temporary buffer so that no destination address needs to be specified; we are using the same type to enable transporting data to a hardware core directly without copying to memory. When a *Medium* message arrives, all payload data is sent through the FIFO that also receives the handler call arguments
- *LongStrided* messages: Long messages which can assemble and distribute non-contiguous data in a two-dimensional layout by specifying a chunk length and stride
- *LongVectored* messages: Long messages which can assemble and distribute non-contiguous data according to explicit size and address specifications

It can be argued that strided/vectored messages are not necessary in the Core API for software and RDMA interconnect solutions, as the functionality can be implemented with Medium type messages and data scattering implemented in the software or hardware handler functionality. However, as our hardware model has a strict separation of concerns (remote memory operations in GAScore, application and handler tasks in embedded CPU or hardware core) and such an approach would put part of the repetitive data handling tasks back into the computation core's hands, we think the establishment of the two non-standard Active Message types is the cleaner solution.

C. Programmable Active Message Sequencer

As the FIFOs connecting the processor to the GAScore are a hardware interface, the CPU can be substituted with a custom processing core as long as the core can reproduce the necessary communication with the GAScore. This can obviously be custom-designed for any processing core, however it likely involves a lot of redundant functionality between otherwise dissimilar computation cores.

To save core designers this repetitive task, we have designed a small application-specific processor, called *Programmable Active Message Sequencer* (PAMS), that is geared towards receiving and generating Active Message parameters and controlling custom core operation. As it is targeted towards a small set of specific functions, it has a much smaller resource footprint and lower latency to react to events than a complete universal processor would.

PAMS functionality includes:

- Setting control signals for the custom hardware or waiting for signals from the hardware
- Waiting for a certain number of specific Active Messages or a certain amount of data to arrive
- Taking the time of message arrival or core completion
- Sending messages off at specific timer values

Based on these features, PAMS code can easily and efficiently implement higher-level functionality like barriers as well as GASNet *Reply*-type functions to enable remote reads. In principle, PAMS code can implement all the essential parts

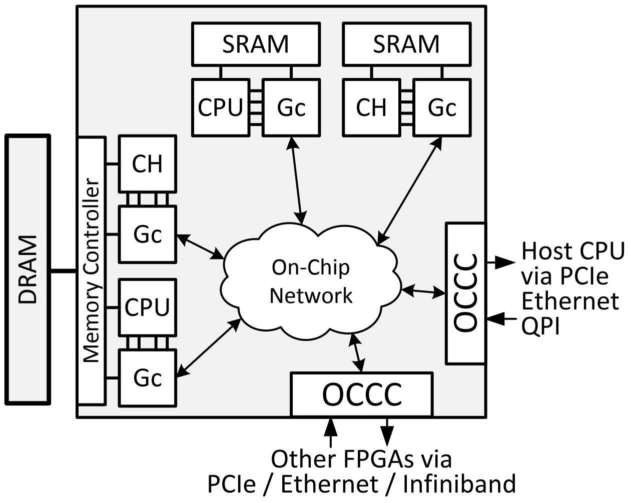
SRAM
SRAM
CPU
Gc
CH
Gc
CH
Gc
CPU
Gc
DRAM
Memory Controller
On-Chip Network
OCCC
Host CPU via PCIe Ethernet QPI
OCCC
Other FPGAs via PCIe / Ethernet / Infiniband

Fig. 4. FPGA with embedded CPUs, custom hardware cores (CH), GAScores (Gc), on-chip and off-chip RAM, and possible off-chip connections (Off-Chip Communication Controller, OCCC)

of the GASNet Extended API, although we have not provided this yet.

An important feature is the capability to re-program each PAMS by sending a specific Active Message of type *Medium* that includes the new code as data payload. This gives great flexibility both for design and debugging purposes and for complex, dynamically changing computation problems. Note that this re-programmability currently only applies to the binary code run by the PAMS itself, not to reconfiguration of the computation core hardware. This is, however, quite feasible as future work.

D. FPGA system overview

Figure 4 demonstrates the different hardware options enabled by the presented components. GAScore can be connected either to embedded CPUs or to custom hardware cores, and both elements can either share on-chip SRAM or, for much larger datasets, an off-chip DRAM accessed through a memory controller. Contrary to this depiction, in most cases all these different combinations will likely not be used concurrently, but rather at different design stages as illustrated in Figure 1 and Section VI. However, the common GASNet interface used in all cases enables all these different combinations to work together. On the borders of the FPGA, we can also see Off-Chip Communication Controllers (OCCC) being connected to the NetIF network. These represent different possibilities to either connect FPGAs together or interface to a host processor, e.g. an x86-64 processor via PCIe or a hardwired ARM core via AXI.

V. THEGASNET SOFTWARE IMPLEMENTATION

A. Compatibility considerations

Building a heterogeneous extension to an existing API, we were faced with the choice to either extend the existing GASNet code base or to implement our own version of the API. We have chosen to do the latter for the following reasons:

- Extending the existing GASNet would have drawn much of our limited resources to compatibility and maintenance work.
- To accommodate heterogeneity, it would be hard and most likely beside the point to stay completely compatible. Instead, we can concentrate on identifying the specific needs of a multi-platform variant, and suggest these for improvement efforts, specifically GASNet-EX [12].
- We were able to keep the footprint of our MicroBlaze Core API implementation small enough to run credible applications code just from on-chip memory, which simplifies the processor system architecture.

A well-understood consequence of this approach is that it is not a realistic possibility to adapt established and productively used PGAS languages to our library in the near future. Instead, we plan our own, heterogeneity-specific high-level library (see Section VIII) which shares many of their properties. Our effort is intended as a technology demonstration and not expected to be adopted for use by the larger community.

B. Differences to the GASNet specification

Our software implementation of the GASNet Core API was written with a focus on reaching compatibility and interoperability between the following platforms:

- Hardware components using the GAScore FIFO interface (and optionally PAMS)
- x86-64 compatible 64-bit processors, multi-threaded, little-endian, running under 64-bit Linux
- ARMv7 compatible 32-bit processors, multi-threaded, little-endian, running under 32-bit Linux
- Xilinx MicroBlaze 32-bit soft processors, single-threaded, little-endian, running bare-metal applications

This included the goal to be able to compile the same C application code, including the same THeGASNet header and calling the same GASNet Core API functions, with *gcc* for all CPU platforms (compiler flags and makefiles can differ)

As a result of these intentions, the following departures from strict compatibility were necessary:

- GASNet API functions use the native pointer type `void*` for both source and destination pointers, as well as for the address and size information in the struct `gasnet_seginfo_t`. As the different CPUs can have native pointers of different sizes (32- or 64-bit), we have defined a 64-bit integer type `voidp64` that has to be used in the segment info and destination address fields and can be casted from and to native pointers as necessary.
- We introduced *strided* and *vectored(scatter/gather)* operations at the Active Message/Core API level. Although they have only been suggested by Bonachea [13] as *put()/get()* variations at the Extended API level, our hardware model suggests specific Core API messages at the GAScore level are necessary and useful.
- As x86-64 and ARM implementation use Linux, but the MicroBlaze runs on a bare-metal runtime platform, specific calls to print and to use the system timers need to be encapsulated in wrappers that are decoded differently for each platform. While these platform-specific definitions are not strictly part of the GASNet API, they have been included as part of our library for reasons of convenience.
- While not technically infeasible, we have decided not to include a shared segment info table in each hardware core for reasons of resource economy. Instead, when communicating with software nodes that don't have a statically defined shared segment address (x86-64 and ARM), hardware cores send an address offset which is added to the shared segment base by the receiving node. We are considering whether this might in fact be a sensible approach all throughout the system, as adding the offset at the destination does not incur a large performance penalty, while table storage grows inconveniently for large-scale systems.

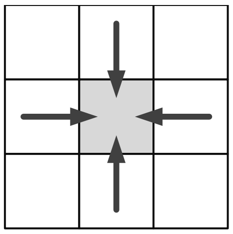
Fig. 5. Jacobi heat transfer stencil operation

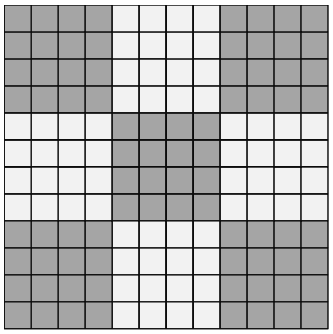
Fig. 6. Node partitioning

VI. Use Case

A. Stencil Computation: Jacobi Heat Transfer

The Jacobi method is an iterative solver for partial differential equations. A practical application for this is to find the steady-state heat distribution on a rectangular slab, e.g. the surface of a chip. The surface can be partitioned into a fine two-dimensional grid of cells. On each iteration, the temperature of a cell is recalculated based on the temperature of the four neighbouring cells to the left, top, bottom and right (also see Figure 5):

$$T_{(t+1),x,y} = \frac{T_{t,(x-1),y} + T_{t,x,(y-1)} + T_{t,x,(y+1)} + T_{t,(x+1),y}}{4}$$

This memory access and computation pattern using spatially local input data is called a *stencil*. It is not just common to iterative equation solvers, but can also be found in many other applications like image filtering. Cellular automata like the popular programming assignment *Conway's Game of Life* [14] are a further example. While *Game of Life* can be computed much more efficiently with hash-based methods, it

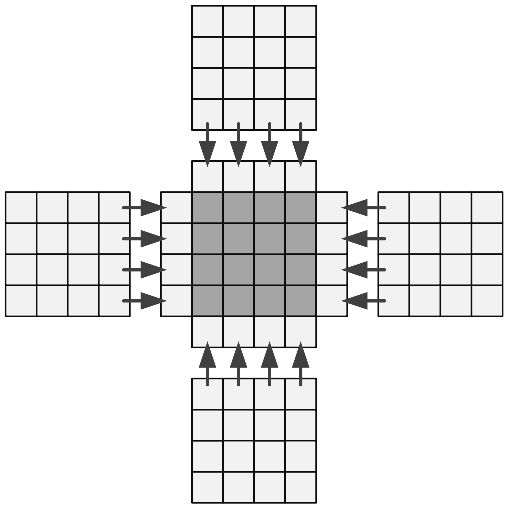

Fig. 7. Node-to-node communication between iterations

is a useful tool for debugging a stencil computation pattern as it gives easily checkable visual output.

Stencil computations are great for parallelization because of their data locality. A larger grid can easily be split up into subgrids (see Figure 6) to compute on separate nodes. It is not embarrassingly parallel in the technical sense, because after each operation neighbour data has to be exchanged between nodes (Figure 7), but the computation-to-communication ratio is high and scales well (the communication grows only with a square root of the problem size, while the computation grows linearly).

As results in Section VII suggest, this application might not be a candidate for large FPGA speedups. However, we think it is a good use case for demonstrating software and hardware THeGASNet implementations because of its regular but non-trivial communication pattern.

B. Test platform

Our test platform for heterogeneous applications, *MapleHoney* (see Figure 8), is a combination of four PC workstations and four Virtex-6 FPGAs on a single BEE4 multi-FPGA board. The PCs each have an Intel Core i5-4440 quadcore processor with 6MB L2 cache, running at 3.1Ghz, and 8GB of DDR3 RAM. The PCs are fully connected through a Gigabit Ethernet switch. The BEE4 FPGA board holds four Xilinx Virtex-6-LX550T FPGAs, each with two 8GB DDR3 memory channels. The FPGAs are connected in a ring topology with a 400MB/s low-latency parallel bus, enabling an inter-FPGA bandwidth equivalent to a duplex 32-bit FSL connection at 100MHz. Each FPGA is connected to one of the PCs by a 4-lane Gen 1 PCIe connection, yielding a net bandwidth of 800MB per direction.

Figure 9 illustrates the internal layout of an FPGA with a THeGASNet configuration. The internal NetIf network connects the external PCIe and FPGA ring interfaces as well as up to eight GASNet nodes with each other. These nodes can be either MicroBlaze processors with an attached GAScore, or custom processing cores with a GAScore. Up to four nodes share a memory channel, whereby the memory controller logically divides the DRAM module up into four

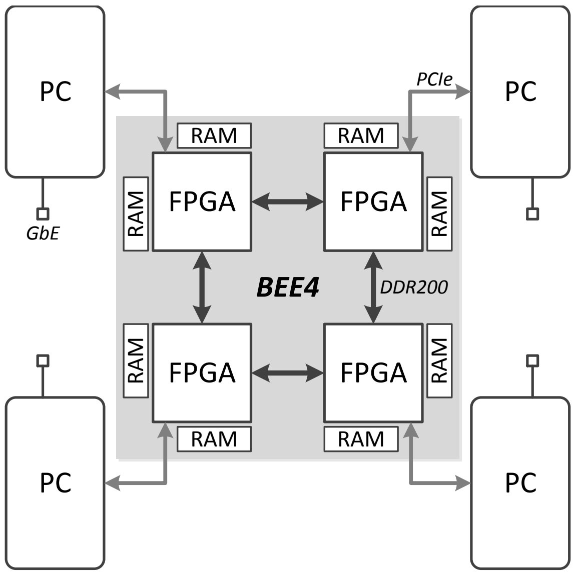

Fig. 8. MapleHoney test platform

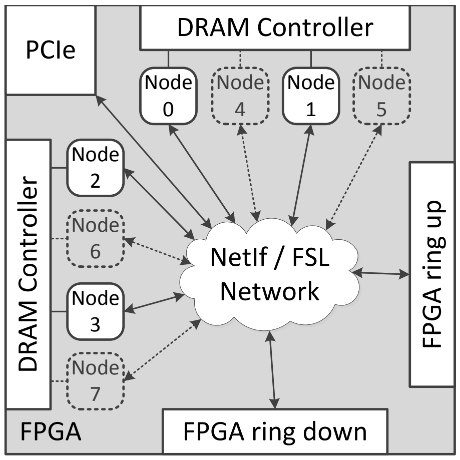

Fig. 9. MapleHoney FPGA layout

separate memories; direct shared use of the same memory section between separate GASNet nodes would not be easily reconcilable with the Active Message model.

In our current benchmarks we are only using four GASNet nodes, so that only two nodes share a memory module. The whole FPGA board therefore hosts 16 GASNet nodes.

C. Implementation Step I: PC code

To design and debug the basic Jacobi application, we wrote code using THeGASNet Core API calls for the x86-64 platform. We are using a 32-bit fixed-point resolution for each temperature marker. Since the stencil application can not be easily implemented *in place*, i.e. by overwriting old values with new ones, we are using a double-buffering approach for the computation: Each alternating iteration reads from the previously written results, and overwrites the read input of the previous iteration.

After each iteration, GASNet *Long* messages write the data in the outermost cells of the local partition to the appropriate fields in the neighbouring partitions (see Figure 7 again). Through handler functions, a node keeps track of the amount of data it has received from neighbouring partitions. If all the expected data has been received, the node continues its computation. Through this "soft barrier" nodes can stay synchronous without the need for global barrier calls.

As a problem size for the four-workstation software implementation, we have determined a square field of 44760*44760 cells, requiring almost 16GB of storage for the double buffering approach at 4 bytes of storage per cell value (Note that each buffer also has to store an "aura" of width one in all four directions as input values for the outermost local cells; these are the data fields that will be filled up by Active Messages from other nodes). Each workstation has therefore a storage requirement of 4GB.

The application is benchmarked by doing five runs of 100 iterations and then averaging the time per iteration. Through benchmarking we have determined that the optimal number of GASNet nodes per workstation is four, making optimal use of the four (non-hyperthreaded) processor cores. The data field is therefore sliced up into 16 GASNet nodes of 11220*11220 cells. The application is started on each workstation with command-line parameters for the complete field size, and uses GASNet information to determine the local partitions, yielding very flexible application code.

D. Implementation Step II: MicroBlaze code

In the spirit of THeGASNet, C application code compiled for the PC should compile without complication for the MicroBlaze. In fact only one adjustment was necessary: The space for the shared segment information table was dynamically reserved on the PC, while the MicroBlaze version statically allocates it on the stack; dynamic allocation on simple embedded systems is generally avoided.

While dynamic specification of command-line parameters (in this case, the data field size) is not possible for the bare-metal MicroBlaze application, the main()-wrapper function that is included when linking the application with THeGASNet can pre-define the *argc/argv* fields based on compile-time parameters.

As we have configured our FPGA design to include four GASNet nodes with a MicroBlaze processor on each FPGA, we end up with a system of 16 MicroBlaze processors. The non-multithreaded MicroBlazes can only run one GASNet node each, so that we end up with the same number of GASNet nodes, and the same amount of data per node, as in the PC system.

The system is benchmarked the same way as the PC system. Obviously performance will be much weaker, as we are replacing a system of powerful superscalar out-of-order processors with the same number of simple in-order embedded processors running at a fraction of the clock speed. However, this is not the point of the code migration. By running the identical application on the FPGA infrastructure, we have opened the door to replacing the processors with hardware computation cores, which just need to show the same communication behaviour as the processors.

As a side note, it is perfectly possible, and has been tested, to run a shared computation between PC and MicroBlaze nodes. Obviously, homogeneous problem sharing like this does not make a lot of practical sense as the MicroBlaze's lower speed is dominating system performance. The intended interoperability works between PC and ARM nodes as well, and is expected to do so between ARM and MicroBlaze.

E. Implementation Step III: Customized hardware computation cores

With the working communication infrastructure for the application in place, a hardware core for the Jacobi stencil computation was designed by us. The existing framework predefines the following port layout for the core:

1) An AXI bus master port, which can burst-read and -write data from the main memory shared with the GAScore. The Jacobi core will read data from one buffer in main memory, and write it back to the second buffer.
2) Duplex FSL connections to the GAScore to communicate Active Message requests and receive handler information about incoming messages. As we are inserting a PAMS instance for the communication handling, the required port reduces to simple control signals to synchronize operation with the PAMS.

The actual implementation of the core is quite straightforward given the computation pattern. The core reads three rows of input into local SRAM buffers with minimal access latency. As this point, the three row buffers can be read out in lockstep, yielding the stencil information for a new row of results to be written back to the main memory. While the new row is computed and written, a fourth row buffer can already read the next row from main memory. After the first row of results has been written, the second and third row, together with the newly read fourth row can be used as input for the stencil computation of the second row of results. The very first input row can at this point be discarded, and the buffer can be refilled with the fifth row of input data. This process repeats until all the newly computed rows have been written.

As SRAM storage on the FPGA is scarce, a single stencil core will not have enough storage capacity for four complete rows. Therefore, the core will read part of each of the first four rows as described before, and so on. The node's data is therefore actually processed in strips or columns as wide as one row buffer is deep.

In a departure from the straightforward code migration described so far, we have determined that the best way to run the system is actually to include a 17th node running software based on the following needs:

- The most convenient way to put the communication code into the different PAMS is to send them the code from a software node (the software node can also conveniently generate the individual addressing pattern for each PAMS, as the PAMS itself does not have an ALU to determine these).
- Letting the PAMS send short messages to the software node after finishing a task is the easiest way to take time for the benchmarking.

TABLE I. RUNTIMES PER JACOBI HEAT TRANSFER ITERATION

Platform	Time(secs)
Software - x86-64	4.32
Software - MicroBlaze	336.13
Hardware - Stencil core	5.83

- A software node is also the simplest way to put the initial computation data into the FPGA system, and retrieve the final data. This last point is as true when using the MicroBlaze processors.

While arguably harming the perfect picture of straightforward migration, this addition of a managing software node actually emphasizes the fact that the system is easily configurable, modifiable, and accommodating in its interoperability of software and hardware components.

VII. RESULT AND CONCLUSIONS

Based on the described benchmarking procedures and problem size, we have determined average runtimes per iteration (from start of one computation to the start of the next computation) as listed in Table I.

As expected, the MicroBlaze software performs much worse than the PC software, especially as the current MicroBlaze configuration does not use a data cache except a one-burst buffer in the memory controller, and therefore incurs several read bursts to external memory for each single cell.

Contrary to our expectations, the hardware solution actually performed 35% worse than the PC software solution. While we expect that both software and hardware solutions have significant room for improvement (our current FPGA memory controller prototype certainly has quite some optimization potential), we do not expect either solution to pull significantly ahead. Therefore, arguably the extra effort for writing a hardware solution does not seem justified here.

Mostly, this is an indication of a poor choice of problem to demonstrate FPGA performance benefits. The Jacobi heat transfer algorithm has a fairly simple computation at its core, and therefore seems to be bounded by memory performance.

Nevertheless, we hopefully made a convincing case for the ease with which a PGAS application using GASNet can be transformed to run on FPGA hardware. For our example, the hardware core was hand-designed and -optimized in VHDL. The current advances in High-Level Synthesis indicate that this part of the design problem can be solved more productively in the future. However, most current HLS tools focus on generating the hardware code to solve an algorithm, and do not focus on push-button solutions for a complete FPGA design. As such, our infrastructure complements these efforts in a promising way: HLS tools can produce the computation core, while THeGASNet provides the framework in which to execute the HLS-generated code.

VIII. FUTURE WORK

As is obvious from the previous sections, we are presenting early results for our recently completed framework. Numerous performance optimizations for memory accesses, network-on-chip infrastructure and GAScore operations are planned.

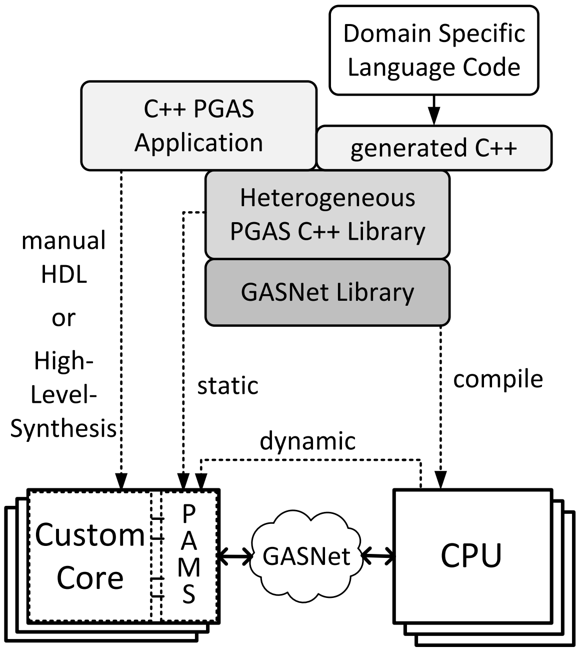
Domain Specific Language Code
C++ PGAS Application
generated C++
Heterogeneous PGAS C++ Library
GASNet Library
manual HDL or High-Level-Synthesis
static
compile
dynamic
Custom Core
PAMS
GASNet
CPU

Fig. 10. Proposed PGAS software stack and code/binary generation

Furthermore, a richer set of benchmarks with a diverse set of computation and communications patterns is necessary for a complete evaluation. We aim to include applications that show interesting heterogeneous characteristics so that CPU and FPGA portions can complement each other.

A. PGAS C++ library for heterogeneous systems

Offering GASNet compatibility for FPGA components at the API level is an important first step, but it does not yet enable significantly higher programming productivity than MPI. To provide that, a higher-level programming facility in the form of a PGAS language or library needs to leverage the benefits that GASNet provides. For CPU platforms many solutions like Global Arrays[15], Unified Parallel C[16], Chapel[17] and X10[18] exist. Instead of adapting one of these tools to our infrastructure, we plan to provide a PGAS C++ library for heterogeneous computation. It will inherit concepts from the established PGAS languages and libraries to enable complex data classes like multi-dimensional arrays, awareness of locality and heterogeneity, configurable data distribution and platform-dependent computation patterns. Many scientific areas have developed Domain-Specific Languages (DSLs) for better modeling of their specific problems, therefore we will enable the library to work as a target runtime for DSL code generation.

A typical toolchain for this concept is illustrated in Figure 10. Either a user-programmed C++ PGAS application or C++ code generated from DSL code will use our heterogeneous C++ library, which employs GASNet as a remote communication layer. The whole application can be compiled to run without custom hardware on host CPUs or on FPGA-based CPUs, with GASNet/GAScore taking care of parallel communication. Customized hardware cores can be manually written or generated by the previously mentioned tools. Our library can generate the necessary PAMS communication code at compile-time. In addition, in the case of changes dependent on control flow, the CPU binary can generate updated PAMS code during runtime.

B. Partial reconfiguration of computation cores

As hinted at in Section IV-C, besides sending PAMS binary code to a node by Active Message, it would also be feasible to reconfigure the computation core hardware with Active Message data through the FPGA's capability for *Partial Reconfiguration* of designated regions. This is simplified by the observation that, as in our current MapleHoney implementation depicted in Figure 9, major infrastructures for memory access, off-chip communication and on-chip network are either completely static between designs or limited in variation (e.g. different number of GASNet nodes/memory clients). Thus, a limited number of base infrastructure bitstreams with reconfigurable computation regions could be re-programmed for different tasks on-the-fly during a GASNet application run. Obviously, this could be neatly integrated into the C++ library discussed above.

IX. Acknowledgements

We would like to express our special gratitude to the diligent reviewers for their helpful recommendations, which we regretfully could only partially honour because of time constraints. Furthermore, we would like to thank the CMC, NSERC and Xilinx for supporting our research.

References

[1] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29*, 8 December 2008. [Online]. Available: http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf

[2] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[3] M. Saldaña, A. Patel, C. Madill, D. Nunes, D. Wang, P. Chow, R. Wittig, H. Styles, and A. Putnam, "MPI as a programming model for high-performance reconfigurable computers," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 4, pp. 22:1–22:29, Nov. 2010.

[4] D. Bonachea, "GASNet Specification, v1.1," University of California Berkeley, Technical Report UCB/CSD-02-1207, October, 2002.

[5] R. Willenberg and P. Chow, "A remote memory access infrastructure for global address space programming models in fpgas," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '13. New York, NY, USA: ACM, 2013, pp. 211–220. [Online]. Available: http://doi.acm.org/10.1145/2435264.2435301

[6] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews, "Enabling a uniform programming model across the software/hardware boundary," in *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, april 2006, pp. 89 –98.

[7] V. Aggarwal, A. D. George, C. Yoon, K. Yalamanchili, and H. Lam, "SHMEM+: A multilevel-PGAS programming model for reconfigurable supercomputing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 3, pp. 26:1–26:24, Aug. 2011.

[8] T. El-Ghazawi, O. Serres, S. Bahra, M. Huang, and E. El-Araby, "Parallel programming of high-performance reconfigurable computing systems with Unified Parallel C," in *Proceedings of Reconfigurable Systems Summer Institute*, 2008.

[9] "Impulse C," http://www.impulseaccelerated.com/.

[10] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz, "Ramp blue: A message-passing manycore system in fpgas," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, Aug 2007, pp. 54–61.

[11] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks - summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 158–165. [Online]. Available: http://doi.acm.org/10.1145/125826.125925

[12] "GASNet-EX collaboration," https://sites.google.com/a/lbl.gov/gasnet-ex-collaboration/.

[13] D. Bonachea, "Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet," Lawrence Berkeley National Lab, Technical Report LBNL-56495, March 2007.

[14] A. H. Dekker, "The game of life: A clean programming tutorial and case study," *SIGPLAN Not.*, vol. 29, no. 9, pp. 91–114, Sep. 1994. [Online]. Available: http://doi.acm.org/10.1145/185009.185032

[15] J. Nieplocha, R. Harrison, and R. Littlefield, "Global arrays: a portable *shared-memory* programming model for distributed memory computers," in *Supercomputing '94. Proceedings*, nov 1994, pp. 340 –349, 816.

[16] UPC Consortium, "Upc language specifications, v1.2," Lawrence Berkeley National Lab, Tech Report LBNL-59208, 2005. [Online]. Available: http://www.gwu.edu/ upc/publications/LBNL-59208.pdf

[17] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The cascade high productivity language," in *in Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 04)*, 2004, pp. 52–60.

[18] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538.