Custom FPGA-based Soft-Processors for Sparse Graph Acceleration

Nachiket Kapre
School of Computer Engineering
Nanyang Technological University, Singapore 639798
Email: nachiket@ieee.org

***Abstract*—**

FPGA-based soft processors customized for operations on sparse graphs can deliver significant performance improvements over conventional organizations (ARMv7 CPUs) for bulk synchronous sparse graph algorithms. We develop a stripped-down soft processor ISA to implement specific repetitive operations on graph nodes and edges that are commonly observed in sparse graph computations. In the processing core, we provide hardware support for rapidly fetching and processing state of local graph nodes and edges through spatial address generators and zero-overhead loop iterators. We interconnect a 2D array of these lightweight processors with a packet-switched network-on-chip to enable fine-grained operand routing along the graph edges and provide custom send/receive instructions in the soft processor. We develop the processor RTL using Vivado High-Level Synthesis and also provide an assembler and compilation flow to configure the processor instruction and data memories. We outperform a Microblaze (100 MHz on Zedboard) and an NIOS-II/f (100 MHz on DE2-115) by ≈6× (single processor design) as well as the ARMv7 dual-core CPU on the Zynq SoCs by as much as 10× on the Xilinx ZC706 board (100 processor design) across a range of matrix datasets.

I. Introduction

Computations on sparse graphs are a challenge for modern multi-core processors due to the irregular nature of memory access involving sparse graphs. Graph problems arise regularly across a wide variety of application domains such as scientific computing (sparse matrix), circuit CAD (netlist), artificial intelligence (semantic knowledge-base), social networking (user connections graphs) among many others. At the heart of these algorithms, we have a common recurring computing pattern involving access to irregularly spaced data items. In these cases, we typically need to repeatedly iterate over nodes and edges of the graph while performing lightweight local computation at each node and/or edge. The core computation can simply be described as loop-oriented operations nested over nodes and edges (see Function graphalg later).

A naïve parallel implementation of graph computations would require distributing subsets of the graph across ISA-style multi-core processors with shared/distributed caches. While this seems straightforward, performance will suffer due to a variety of factors. Memory traversal over adjacency lists requires pointer arithmetic and multiple levels of indirection to fetch the required data. When attempting to access data on other cores, the shared memory structure imposes a performance penalty that can be severe for scattered graphs.

In contrast, customized application-specific graph processors with parallel scratchpads and fully-customizable FPGA logic offer an interesting alternative. In these custom organizations, we can arrange for the sparse graph data to reside in distributed fast, high bandwidth FPGA on-chip memories that easily exceed the on-chip cache bandwidth of conventional processors by as much as 10–100×. For processing large graphs, we envision scaling the design across a multi-FPGA setup as discussed in [7] and demonstrated in [14], [15]. In this paper, we focus on the design and engineering of the application-specific soft processor that can be tiled across such a multi-FPGA setup. The sharing of data along graph edges can be implemented directly using wires in the FPGA logic with a custom network-on-chip for orchestrating sharing of the routing resources. When designing custom graph processors on FPGAs, we may choose to implement a fully customized spatial processing datapath from the bottom-up for every graph algorithm like those in [7], [17]. While this may deliver the best performance, the design process will be tedious and suffer the usual long development and compilation cycles of a typical FPGA design flow. Alternatively, we can use off-the-shelf soft processors such as the Altera NIOS-II or the Xilinx Microblaze and tile them instead. However, these soft processors will deliver poor performance due to their slow implementations, excess hardware and lack of deep customization hooks for performance tuning. Hence, we propose GraphSoC, a lightweight soft processor that allows faster design composition of graph algorithms, quicker implementation flow on the FPGA, while delivering high performance.

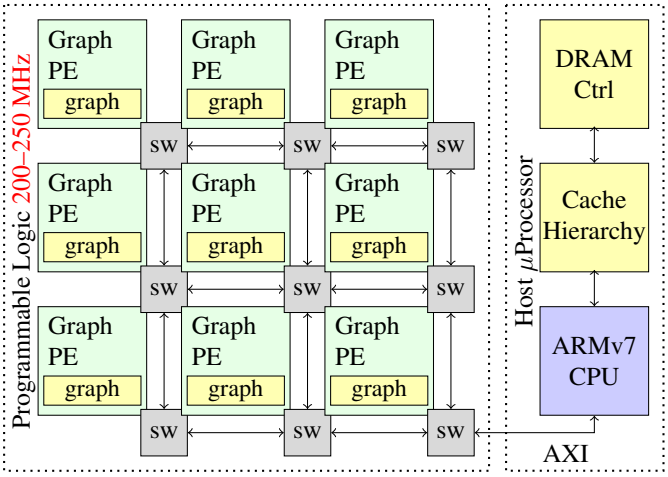


Fig. 1: GraphSoC Accelerator Architecture

The key contributions of this paper include:

Function graphalg(node_state, edge_state)

```
   /* process all nodes                                                    */
1  foreach n = nodes in graph do
   |  /* evalute all input edges                                          */
2  |  foreach ie = input edges of n do
3  |  |  node_state(n) = f(node_state(n), edge_state(ie));
   |  /* evaluate all output edges                                        */
4  |  foreach oe = output edge of n do
5  |  |  edge_state(oe) = g(node_state(n));
```

- Development of the GraphSoC custom soft processor for accelerating graph algorithms in hardware.
- Design of a compilation framework based on Vivado High-Level synthesis for generating specialized graph processors for the Zedboard (25 PE), and ZC706 (100 PE) platforms.
- Quantitative comparison of the performance across a range of datasets when comparing ARMv7 CPU, Xilinx Microblaze and Altera NIOS-II/f with GraphSoC.

II. Background

A. Bulk-Synchronous Parallel Model

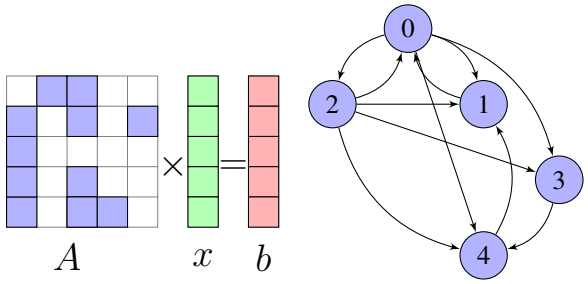
In this paper, we explore parallel graph algorithms that fit the Bulk Synchronous Parallel (BSP) paradigm. The BSP compute model [21], [22] is well-suited for describing parallel graph algorithms for FPGA system architectures [12], [8], [17], [7], [6], [2]. The execution of the graph algorithm is organized as a sequence of steps where the steps are logically separated by a global barrier. In each step, the PEs perform parallel, concurrent operations on nodes of a graph data structure where all nodes send and receive messages from their corresponding neighbors. Once the messages reach their destinations, each node performs a local summarization operation on all input edges. This compute model is applicable to graphs that do not change their topological structure in the midst of the execution flow. While this model might seem specific, it admits parallel descriptions of a wide variety of parallel computations such as sparse matrix-vector multiply, contextual reasoning, belief propagation, all-pairs shortest path search, betweenness centrality among many others. The basic computation can be understood as the nested loops shown in Function graphalg where function f and g operate on the incoming and outgoing edges of a node respectively. In the parallel BSP implementation, we can rewrite these loops to expose node-level concurrency as shown in Function parallel_gstepalg. Here, we split the function f into three specialized graph operations $receive$, $accum$ and $update$ while we represent function g using the $send$ operation. This parameterization allows the processor to be customized for different graph algorithms while simplifying hardware assembly. Additionally, it helps us isolate the local computations on the nodes (f) from memory or communication operations (g) for scheduling freedom. The general description that explains this four-function programming model for graphs has been covered in depth in [7], [6].

B. Sparse Matrix-Vector Multiply Example

Iterative Sparse Matrix-Vector Multiply (SpMV) is the dominant computational kernel in several numerical routines (including integer-oriented cryptanalysis computations). In each iteration a set of dot products between the vector and

Function parallel_gstepalg(node_state, edge_state)

```
   /* chunked parallel for                                                 */
1  foreach n = nodes in graph do
   |  /* pipelined evaluation/summarization                               */
2  |  foreach ie = input edges of n do
3  |  |  recv = receive(edge_state(ie), message(ie));
4  |  |  acc = accum(acc, recv);
5  |  node_state(n) = update(node_state(n), accum);
   |  /* dependencies over packet-switched network                        */
6  |  foreach oe = output edge of n do
7  |  |  message(oe) = send(node_state(n));
```



(a) Sparse Matrix (b) Sparse Graph

Fig. 2: Sparse Matrix as a Sparse Graph

matrix rows is performed to calculate new values for the vector to be used in the next iteration. We can represent this computation as a graph where nodes represent matrix rows and edges represent the communication of the new vector values. In Figure 2, we show how to translate a sparse matrix to a graph. We shade the non-zero entries in the matrix example. Each row (and vector element) corresponds to a node in the graph and each non-zero location corresponds to an edge from the respective vector element node. The graph captures the sparse communication structure inherent in the dot-product expression. In each iteration, messages must be sent along all edges; these edges are multicast as each vector entry must be sent to each row graph node with a non-zero coefficient associated with the vector position. We can re-express the function in the BSP model by defining the four functions as shown in Table I. We use SpMV (streaming multiply-accumulate datapath) to quantify performance on our architecture.

Function	Semantics	Equation
$receive$	multiply with A[i,j]	$temp = A[i,j] \times x[j]$
$accum$	sum the products A[i,j]*x[j]	$acc = acc + temp$
$update$	write b[i] result of accum	$b[i] = acc$
$send$	simply copy x[j] into packet	$x[j] = b[i]$

TABLE I: Sparse Matrix-Vector Multiply example

III. GraphSoC Soft-Processor Design

As shown earlier in Figure 1, we organize our parallel FPGA hardware as a bidirectional 2D-mesh of graph processors supported by a host CPU (ARMv7 CPU) that manages to runtime and device driver support. We choose the Zedboard, and ZC706 boards for prototyping the soft processor core in our current design and evaluate performance against the host ARMv7 CPUs. While we choose the Zynq boards for

prototyping the soft processor core in our current design, we expect to build higher-performance systems in the near future by (1) either scaling up the 2D mesh to larger system sizes possible on denser FPGA platforms, or (2) tiling multiple Zynq SoCs together [14], [15].

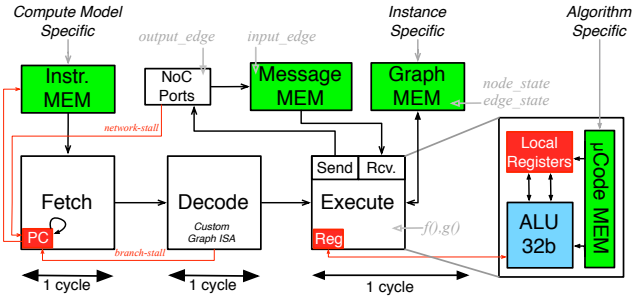


Fig. 3: GraphSoC Processor Pipeline

A. Design Principles

Ultimately, our soft processor implements the pseudocode shown in Function parallel_gstepalg. The core functions f and g in the pseudocode can be implemented as spatial datapaths with the control-flow for the loops implemented as state machines, we prefer a more general and re-programmable approach using a processor-inspired organization. We show a high-level block diagram of our proposed GraphSoC soft processor in Figure 3. It is a 3-stage processor pipeline with customization of instructions to support graph node and edge operations, streamlining of memory operations and other enhancements to support NoC communication. A single FPGA can fit multiple instances of the processor tile interconnected with a custom flit-level NoC. All graph data is stored in on-chip Block RAMs for fast local access. Larger graphs can be partitioned into sub-graphs and loaded one-by-one or split across multiple chips. By avoiding access to large graphs stored in off-chip DRAM, we are able to fully exploit the higher on-chip Block RAM bandwidth available on modern FPGAs (see Section VI-A).

When choosing the soft processor microarchitecture, we considered the use of generalized embedded ISA-based soft processors (*e.g.* Microblaze, NIOS) but found them severely underpowered. Our experiments show a performance gap of as much as 6× (see results later in Section V) when using graph-specific customizations instead of using off-the-shelf soft processors. Soft processors such as iDEA [5] (DSP friendly design) and Octavo [11] (BRAM-friendly design) are equally unsuitable for significant custom instruction augmentation without a complete overhaul.

We now discuss specific characteristics of our design:

- **Graph Algorithm Specialization**: The basic source of specialization in the architecture is the ability to customize the `Execute` stage for various sparse graph algorithms. Thus, our ISA directly supports four types of custom instructions for the `send`, `receive`, `accum` and `update` operations that can be modified for each graph algorithm. These are implemented as high-throughput μcoded datapaths for easy reprogrammability as shown in Figure 4. Thus, to run a new graph algorithm on GraphSoC, we only need to swap in different implementations for these four instructions.

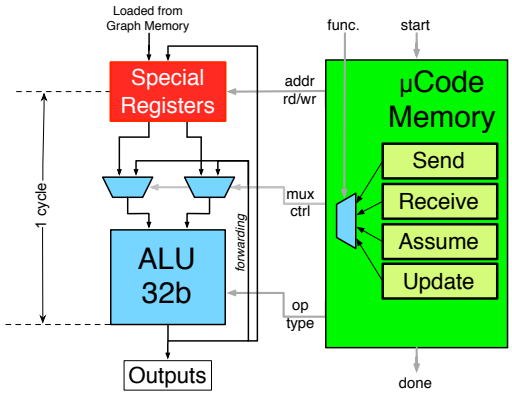


Fig. 4: Block Diagram of the Datapath

- **Graph Memory Optimization**: Since the bulk of the memory operations in the computation are to irregular graph structures, we use a CSR-inspired (compressed sparse row [18]) storage format to support fast access when looping over the graph structure in hardware. We show the memory layout and the communication format for the NoC in Figure 5. We restrict all memory accesses to be in terms of virtual node and edge indices and convert them into physical addresses directly in hardware with dedicated address generators as shown in Figure 6. Thus direct physical address access to the data memory is prohibited. This saves us dozens of instructions per access that a general-purpose ISA would have required.

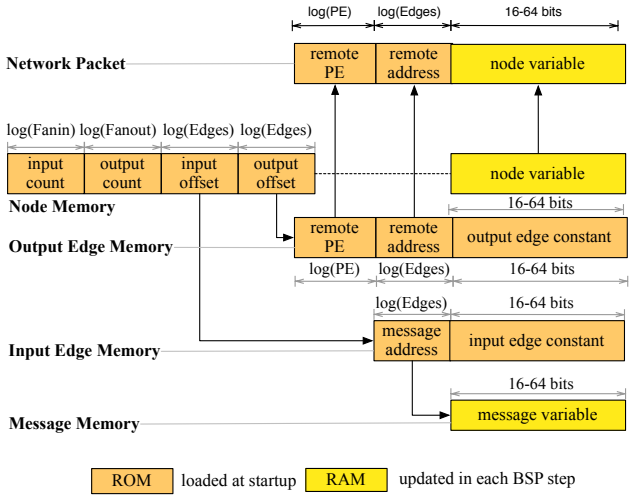


Fig. 5: Memory and Communication Format

- **Special Registers** We do not need a general-purpose register file and dedicate special purpose registers for holding node and edges information instead. We add associated instructions for directly manipulating those registers simplifying the implementation of loads and stores. We also add loop count registers to support zero-overhead looping.
- **Communication Support**: The design of the NoCs on FPGAs is a well-studied topic [9],[1]. We implement the Dimension-Ordered Routing (DOR) algorithm [16] that is simplest to realize in hardware and widely used in NoC designs. In the soft processor, we add hardware support

for (1) non-blocking message receipts where messages are written directly to a dedicated message memory, and (2) blocking message sends that react to network state when incrementing the program counter (stall in Figure 3).

- **Looping and Branching**: Analysis of Function parallel_gstepalg reveals repetitive multi-instruction operations like updates to loop variables and dereferencing the graph pointers to nodes and edges. Consequently, we provide hardware support for loop count registers connected to spatial, pipelined address generators (See Figure 6) to operate in a single cycle. They are similar to zero-overhead loops in DSPs [20], MXP [19], and Octavo [11] soft processors. This optimization saves multiple cycles of instructions on a general-purpose ISA. We also add special branch instructions that access only those loop variables for low-overhead looping. We pack the branch delay slots with useful work and propagate a kill signal across the pipeline stages of the processor when the branch goes the other way just like a normal pipeline stall.

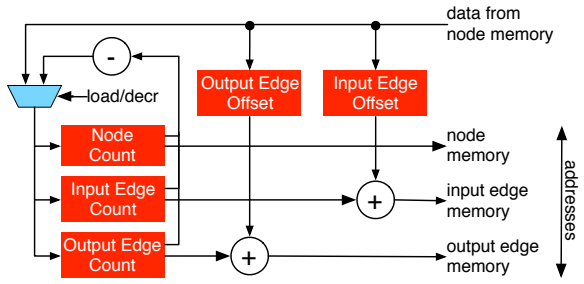


Fig. 6: Design of Address Generator

- **Instruction Fusion**: When we profiled our graph computation during system design phase, we identified certain independent instruction pairs that occur repeatedly in the node and edge loops. This is particularly true for overlapped memory loads from the graph memory blocks and scheduling instructions during variable-latency blocking NoC operations. For example, decrementing the loop counters, NoC send/receive and graph memory loads were the most commonly occurring instruction sequences without dependencies. We show our instruction set (including these fused instructions) in Table II.

B. Assembling and Executing Code on the Soft-Processor

We assemble the RTL for the soft processor by composing (1) logic design of the processor pipelines, (2) instruction memory contents, (3) μcode for the graph algorithm functions, and (3) data memory contents (graph memory) along with a template for the 2D NoC. The underlying processor pipeline structure is fixed and only needs to be compiled through the time-consuming FPGA CAD flow once. Even the instruction memory contents are fixed as the loop-oriented code in Function parallel_gstepalg remains unchanged across graph algorithms, and only needs to be parameterized to handle the subgraph size (*i.e.* number of nodes allocated to the soft-processor). However, our assembler generates μcode on a per-graph application basis which is loaded during the runtime bootstrapping phase once. The structural fields in the graph structure, shown in dark orange in Figure 5, are unique to each graph instance and must be generated separately for each execution even for the same graph algorithm. However, like the μcode memory, it also needs to be loaded into the data memory during runtime bootstrapping at the start. This is because for iterative BSP computations, the structure remains fixed. The only values that change in each BSP iteration are the ones marked in lighter yellow in Figure 5; the state at the nodes and edges. Processor execution can start once logic and memory structures are in place. The loop counter loads the total node count in this processor, proceeds to fetch the entry for the first node from the `Node Memory`. This allows us to load the edge counts, offset and state registers all in a single pipeline cycle. The inner loops over input edges is then processed by decrementing the input counter until it turns 0. The specific entry corresponding to that edge is dereferenced through the hardware address calculator by using the input offset to compute the address to the `Input Edge Memory`. The corresponding message received on that edge is then read and processed. At this stage, the `RCV`, `ACC` operations can proceed as their inputs are available; namely the node state, input edges constant and the input message on that edge. Once all input edges of the node have been processed, the `UPD` performs a state writeback to the `Node Memory`. A similar procedure applies to the output edges with the exception that the execution is trickier due to the blocking nature of the NoC `SND` operation. Once the loop is processed, the computation terminates or the next BSP iteration is launched.

TABLE II: GraphSoC Instruction Set

Instruction	Description
Graph Algorithm Node and Edge Operations	
SND	process node_state to generate packet
RCV	receive edge data from message_memory
ACCU	perform accumulation on input edges of node
UPD	update node state with accumulation result
SAR #imm	initialize accumulation register to imm
Looping and Branching Support	
DC	decrement count by 1
B #lbl	Branch to lbl if count=0
BNZ #lbl	Branch to lbl if count!=0
NOP	No operation
HALT	Halt processor
Graph Memory Operations	
LC	load count to register
LS	load state to register
LMSG	load message into register
Fused Instructions	
DC + SND	decrement count by 1 and launch packet
DC + LS + LMSG	decrement count by 1 and load state, and message

IV. Methodology

In this section, we describe our programming methodology and experimental framework for characterizing the resource utilization of the processor and quantifying its performance.

A. Hardware Engineering

We run our experiments on 32b ARMv7 Ubuntu with suitable Xillybus drivers. For our software baseline (hard processors), we compile the graph algorithm on the ARMv7 32b 667 MHz CPU with `g++ 4.6.3`, with the `-O3` flag (includes NEON optimization for ARMv7). For our soft processor

Name	LUTs	FFs	BRAMs (18KB)	DSP48	Clock (ns)
Fetch	35	24	0	0	2.6
Imem	24	9	0	0	2.9
Decode	2	43	0	0	2.2
Execute	437	305	9	1	4.3
Processor	974	551	9	1	4.3
(%)	1%	1%	3%	0.5%	-
Switch	1882	1076	0	0	4
(%)	2%	2%	0%	0%	-

TABLE III: Resource Utilization of the GraphSoC (Zedboard)

baseline comparison, we compile code for the NIOS-II/f (DE2-115) using `nios2-gcc` with the `-O3` switch and also target the Microblaze (Zedboard) using the `mb-gcc` compiler with `-O3` switch. We express the synthesizable GraphSoC processor functionality in high-level C++ for the individual stages and quantify their implementation metrics (area, frequency, latency, initiation interval). We use the Vivado HLS compiler `v2013.4` for generating RTL. We supply synthesis constraints and directives along with memory resource hints to pack data into sparse but abundant LUT RAMs (for switches) or dense but precious Block RAM resources (for graph memory in PE). We target and achieve an initiation interval of 1 and a system frequency of 200 MHz (Zedboard) and 250 MHz (ZC706) allowing fully-pipelined operation. We support a per-PE node count of 1K and edge count of 2K to fit the Zynq BRAM capacity. Instruction fusion increases overall LUT utilization slightly by $\leq$1% with no impact on delay. In Table III, we tabulate the resource utilization of the different pipeline stages and the sizes of the 2D systems in Table V. We illustrate our processor generation and programming flow in Figure 7 and describe the building blocks below:

- *PE RTL*: The processor pipelines are described in C++ and translated into RTL using High-Level Synthesis. Using Boost pre-processor parameterization [10], we are able to generate multiple instances of the processor to build 2D meshes of required dimensions.
- *Instruction Memory and Execute Stage*: We specify the individual algorithms using C++ API calls which are then compiled to target our processor. The graph developer must supply descriptions of the four graph functions `send`, `receive`, `update` and `accum`. We develop a simple compilation flow based on GIMPLE [13] and our own custom assembler that generates the μcode from these specifications.
- *Graph Memory*: We use a Boost graph library based flexible representation in our runtime to manage the parsing and partitioning of the graph structures. We use the PaToH [4] partitioner to distribute the graph across the PEs to minimize bisection bandwidth. This is an optional one-time task that can be performed once for each graph and is easily amortized (<1s) by iterative evaluation.

Benchmark	Nodes	Edges
add20	2395	17319
bomhof_circuit_2	4510	21199
bomhof_circuit_1	2624	35823
bomhof_circuit_3	12127	48137
simucad_ram2k	4875	71940
hamm_memplus	17758	126150

TABLE IV: Sparse Matrix Vector Multiply Dataset

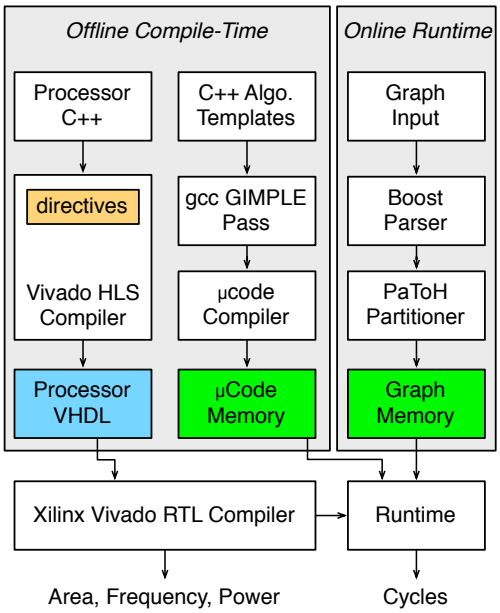


Fig. 7: The GraphSoC Compilation Flow

B. Sparse Matrix Vector Multiply Benchmark

We characterize performance scaling trends on a set of graph benchmark for the sparse matrix-vector multiply graph kernel. We use matrices from Matrix Market [3] library. The graph dataset capture varying structural characteristics that exhibit unique performance trends tabulated in Table IV. We verify functional correctness of our execution results by comparing the node state in the graph at program termination with the sequential reference baseline.

V. Evaluation

In this section, we present the performance results and scaling capabilities of GraphSoC and analyze performance trends and bottlenecks.

How does the GraphSoC compare against other off-the-shelf soft processors? One way to parallelize a sparse graph problem on FPGAs is across existing off-the-shelf soft core processors. In Figure 8, we show a representative result of time taken for the `add20` dataset across a variety of embedded SoC platforms (one processor only). As we would expect, the NIOS-II and the Microblaze run 5–6$\times$ slower than GraphSoC. The 667 MHz ARMv7+NEON is about 3$\times$ faster than the 1 PE GraphSoc implementation as expected due to faster clock frequency. These results highlight the clear benefits of customization for the algorithm in hardware, but still justify the need for an array of such lightweight customized processors to make the parallel design competitive with conventional CPUs.

How does the Graph SoC compare against an equivalent optimized software implementation on conventional proces-

Board	FPGA	LUTs	FFs	BRAMs	DSP48	System
Zedboard	XC7Z020	53K	106K	140	220	5$\times$5
ZC706	XC7Z045	218K	437K	545	900	10$\times$10

TABLE V: FPGA Capacities and GraphSoC system sizes on Zynq-based boards

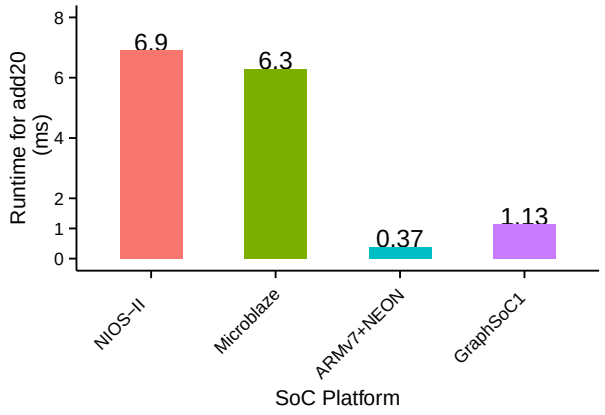


Fig. 8: Comparing Runtime of various Soft Processors for `add20` (one processor case)

sors? In Figure 9, we compare the performance of the ARMv7 CPU + NEON implementation against the runtime of the 25 PE (Zedboard) and 100 PE (ZC706) Graph SoC. We observe speedups in all problem instances (including small datasets) with peak speedups of 5.5× (Zedboard) and 10.5× (ZC706). The variation in speedup is due to imbalances in the distributed workload across the different processors and communication locality and sequentialization bottlenecks due to high-fanout nets. In [14], we show how to cluster multiple low-cost Zedboards and exceed the performance and energy-efficiency of a server-class Intel E5-1650 x86 processor with 16–32 Zedboards.

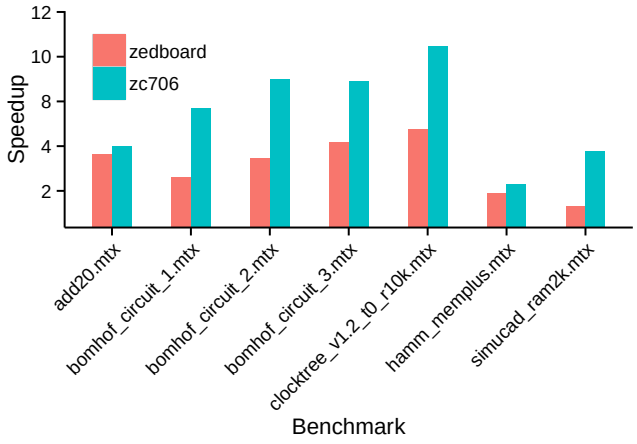


Fig. 9: Comparing GraphSoC (Zedboard and ZC706) with Conventional Processor (ARMv7)

How do we account for the speedups of the GraphSoC compared to other soft and hard processors? To quantify the benefits of the different optimizations to the ISA and associated supporting hardware in our design, we profile the different datasets at 1 PE by successively disabling various optimizations and recording resulting cycle counts. We report our observations in Figure 10. Parallelizing across multiple GraphSoC PEs gives us 10–20× speedup over a single GraphSoC PE (see Figure 11 later), so we focus on the 1 PE scenario. Coalesced graph memory accesses to load/store specialized node and edge state registers save 3–4 loads/stores

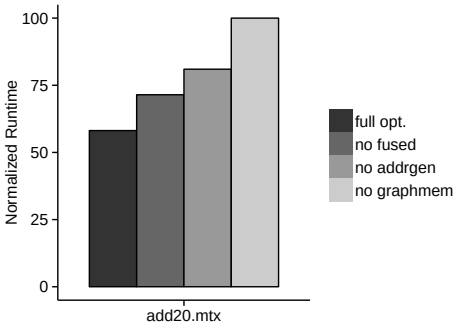


Fig. 10: Impact of Customization of Instructions (at 1 PE)

per access thereby accounting for a substantial 20% saving in cycles. Custom address generators save 1–2 cycles of loads and addition calculations per node and edge operation which adds up to roughly 10% savings in cycle count. Careful fusion of overlapping instructions further saves 10% more. Taken together, these optimizations add up to roughly 45% savings (almost halving runtime of the computation). The simpler hardware design of the GraphSoC eschews the hardware complexity of supporting a NIOS-II/f and Microblaze ISAs, complete register files, caches and other peripherals. This helps keep our design lean and fast at >200 MHz compared to the NIOS-II/f (100 MHz) and Microblaze (110 MHz) respectively. This accounts for another 2× in performance. However, when compared to the ARMv7 NEON accelerator, a single PE GraphSoC designs runs about 2× slower. The 667 MHz 2-lane 32b NEON engines have a much higher 32b peak parallel processing potential compared to the 1 PE 200 MHz 32b datapaths of our soft processor. Neither of these are able to achieve their peak potential due to irregularity of memory accesses, but the GraphSoc actually performs better than expected due to simpler memory accesses.

What are the performance scaling trends for the GraphSoC as we increase PE count? In Figure 11, we quantify the impact of varying PE counts on overall graph algorithm performance. We observe close to linear scaling for virtually all our datasets. The `bomhof_circuit_3` and `hamm_memplus` datasets show early onset of performance saturation among our datasets. For the `add20` dataset, we see a slowdown bump at 16 PEs due to these imbalances in workload distribution at that PE count. :w

What are the fundamental architectural bottlenecks in the soft processor? How can we overcome them? While our system delivers speedups, scalability is somewhat constrained as we observe the mere 2× improvement in performance on ZC706 board that is 4× larger and 25% faster than the Zedboard. Analysis of the bottlenecks, reveals that it should be possible to push performance further. In Figure 12, we show the result of profiling. As we increase PE count, the unaccounted fraction of total cycles (less than 100%) indicate misaligned processor halts due to workload imbalance (note the dip at 16 PEs corresponds to the bump previously noted in Figure 11). Beyond hardware modifications, we expect significant improvements to be possible through graph pre-processing in software such as

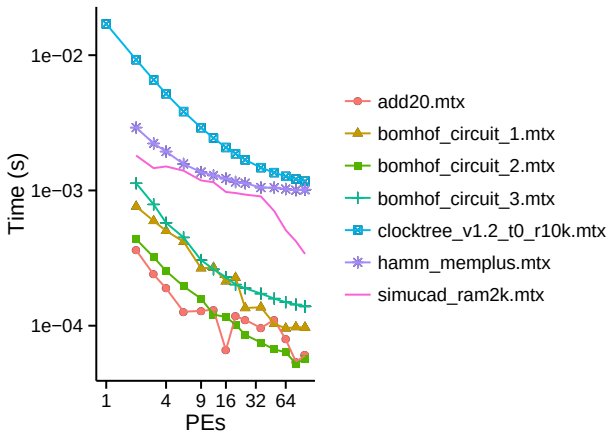


Fig. 11: Impact of PE Scaling on Runtime (ZC706)

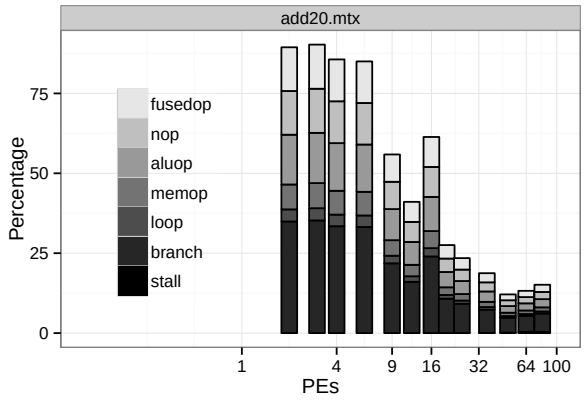


Fig. 12: Averaged Runtime Breakdown (%) for different instructions for `add20` dataset

fanout decomposition, fanin reassociation and better locality-aware placement.

We also observe that the bulk of the dynamic instruction cycles are spent in *branch*. However only a small portion of these are wasted cycles due to branch delay slot usage. These wasted cycles are the *nop* counts and can be further optimized through careful instruction fusion.

We quantify memory and network efficiency by counting the number of useful transaction on the network and memory ports during execution. As the network interactions are relatively infrequent (approximately injecting a packet every 15–20 cycles), network stalls have virtually no impact on performance. At larger system sizes on denser FPGAs we do expect network effects to matter. In Figure 13(a), we observe up to 6–12% network port utilization across our datasets. The processor uses on-chip graph memory bandwidth at an efficiency of 15–20% as seen in Figure 13(b)) which matches the number of memory operations issued by the GraphSoC pipelines. We currently process the computation and communication phases of BSP algorithm in sequential manner due to a single fetch-decode-execute pipeline. We could potentially remedy this by creating two separate pipelines with separate sets of Program Counters, Fetch, Decode and Execute stages that share the same graph memory. In this hypothetical implementation, both the compute and communication phases can proceed in overlapped fashion and approach the performance of the spatial graph datapaths.

VI. Related Work

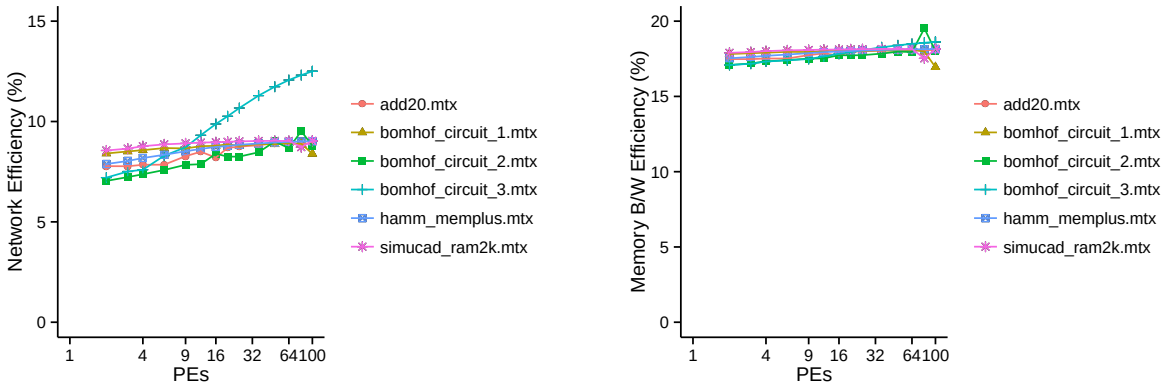
Our framework is inspired from a variety of graph processing frameworks [7], [12], [8], [17], [2]. **GraphLab** [12] is a C++-based graph abstraction for machine learning applications developed for multi-core and cloud platforms with no FPGA support yet. **Green-Marl** [8] is another domain-specific language with a high-performance C++ backend for graph analysis algorithms also missing FPGA support. The **GraphStep** [7] is one of the earliest system architectures for sparse, irregular graph processing algorithms specifically aimed at FPGAs. GraphStep hardware was composed from hand-written low-level VHDL implementations of customized hardware datapaths without any automated compiler/code-generator support. **GraphGen** [17] is a modern FPGA framework that supports automated composition of sparse graph accelerators on FPGA hardware (ML605/DE4 boards). Like Graphstep, there is no compiler for generating graph datapaths but they can be supplied as templated VHDL or Verilog. There is an automated plumbing system that directly interfaces with the DRAM. The top frequency of their spatial designs were limited to 100 MHz (ML605 board) and 150 MHz (DE4 board). Importantly, the graph data is streamed over the DRAM interface without exploiting locality through an NoC limiting performance to DRAM speeds. In [2], the authors investigate the parallelization of shortest-path routing on the Maxeler platform, but are similarly restricted to 2× speedup over multi-core CPUs due to the reliance on DRAM interface bandwidth.

A. Handling large graphs

In contrast to these designs, GraphSoC handles large graph structures by scaling to multiple SoC boards such as the one demonstrated in [14], [15] while keeping the graph data entirely onchip. Our approach allows us to scale out to multiple SoC boards while keeping data resident on-chip to exploit the 10–100× faster on-chip memory and NoC bandwidths for supporting sparse graph communication. Our compute organization exploits high-throughput on-chip memory bandwidth spread across dozens of cheap, low-power Zynq SoCs instead of suffering the limits of the off-chip DRAM interface bandwidths. We prototype a (1) 32-node Zedboard cluster [14] that doubles the performance of a server-class Intel x86 CPU at identical energy efficiency, and (2) 16-Microzedboard cluster [15] delivers identical performance at 30% more energy efficiency for sparse graph processing workloads. The key contribution in [14], [15] is the design and development of an optimized message-passing MPI library layer for sparse graph communication between FPGA accelerators over Ethernet.

VII. Conclusions

Our FPGA-based GraphSoC soft processor is able to outperform the Microblaze (100 MHz Zedboard) and NIOS-II/f (100 MHz DE2-115) by ≈6× when considering a single processor design. We beat the ARMv7 CPU by up to an order of magnitude when using the ZC706 FPGA (100-processor design) across a range of matrix datasets. We demonstrate scalability up to 100 PEs and are able to deliver these



(a) Network Port Efficiency

(b) Memory B/W Efficiency

Fig. 13: Understanding Graph SoC Efficiency

speedups due to customized graph memory access operations, specialized address generators, zero-overhead loop iterators and select instruction fusion optimizations. Our HLS-based graph programming API will allow developers to write new graph algorithms for our GraphSoC, beyond the sparse matrix-vector multiply benchmark. As shown in [14], [15], we can exceed the performance and energy efficiency of a server-class, multi-core x86 processor when using a cluster of 16–32 Zedboards or 16 Microzedboards.

REFERENCES

[1] M. S. Abdelfattah and V. Betz. The power of communication: Energy-efficient NOCS for FPGAS. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, 2013.

[2] B. Betkaoui, Y. Wang, D. Thomas, and W. Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 8–15, July 2012.

[3] R. F. Boisvert, R. Pozo, K. Remington, R. Barrett, and JJ. The Matrix Market: A web resource for test matrix collections. *Quality of Numerical Software: Assessment and Enhancement*, pages 125–137, 1997.

[4] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *Parallel and Distributed Systems, IEEE Transactions on*, 10(7):673–693, 1999.

[5] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell. iDEA: A DSP block based FPGA soft processor. *FPT*, 2012.

[6] M. deLorimier, N. Kapre, N. Mehta, and A. DeHon. Spatial hardware implementation for sparse graph algorithms in GraphStep. *ACM Transactions on Autonomous and Adaptive Systems*, 6(3):1–20, Sept. 2011.

[7] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. J. Knight, and A. DeHon. GraphStep: A system architecture for sparse-graph algorithms. In *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*. IEEE, IEEE Computer Society, 2006.

[8] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *ASPLOS '12: Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ACM Request Permissions, Mar. 2012.

[9] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon. Packet switched vs. time multiplexed FPGA overlay networks. In *Proc. 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 205–216. IEEE, 2006.

[10] V. Karvonen and P. Mensonides. The Boost Preprocessor library. Technical report, 2001.

[11] C. E. LaForest and J. G. Steffan. Octavo: An FPGA-Centric Processor Family . In *the ACM/SIGDA international symposium*, pages 219–228, New York, New York, USA, 2012. ACM Press.

[12] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence*. VLDB Endowment, 2010.

[13] J. Merrill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–179, 2003.

[14] P. Moorthy and N. Kapre. Zedwulf: Power-Performance Tradeoffs of a 32-node Zynq SoC cluster. In *Field-Programmable Custom Computing Machines, 2015. FCCM'15. 23rd Annual IEEE Symposium on*. IEEE, IEEE Computer Society, 2015.

[15] P. Moorthy, Siddhartha, and N. Kapre. A Case for Embedded FPGA-based SoCs in Energy-Efficient Acceleration of Graph Problems. In *Supercomputing Frontiers 2015*. A*Star Singapore, 2015.

[16] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, 1993.

[17] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, 2014.

[18] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report RIACS-90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, 1990.

[19] A. Severance and G. Lemieux. VENICE: A compact vector processor for FPGA applications. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 245–245. IEEE, 2012.

[20] G.-R. Uh, Y. Wang, D. Whalley, S. Jinturkar, C. Burns, and V. Cao. Effective Exploitation of a Zero Overhead Loop Buffer. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 10–19, New York, NY, USA, Jan. 2015. ACM.

[21] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), Aug. 1990.

[22] L. G. Valiant. Why BSP computers? [bulk-synchronous parallel computers]. In *Parallel Processing Symposium, 1993., Proceedings of Seventh International*, pages 2–5, 1993.