# Guppy: A GPU-like Soft-Core Processor

Abdullah Al-Dujaili [1], Florian Deragisch [2], Andrei Hagiescu [3], and Weng-Fai Wong [3]

[1] *Nanyang Technological University, Singapore*
[2] *ETH Zürich, Switzerland*
[3] *National University of Singapore, Singapore*
Corresponding author: wongwf@nus.edu.sg

***Abstract*—The popularity of GPU programming languages that explicitly express thread-level parallelism leads to the question of whether they can also be used for programming reconfigurable accelerators. This paper describes Guppy, a GPU-like softcore processor based on the in-order LEON3 core. Our long-term vision is to have a unified programming paradigm for accelerators - regardless of whether they are FPGA or GPU based. While others have explored this from a high level hardware synthesis perspective, we chose to adopt the approach of a parametrically reconfigurable softcore. We will discuss the main architecture features of Guppy, compare its performance to the original core. Our design has been synthesized on a Xilinx Virtex 5 FPGA.**

## I. Introduction

Reconfigurable hardware accelerators have shown that they can significantly improve performance of many applications. However, programming for hardware accelerators have proven to be difficult, and remains one major hurdle to their widespread adoption. The main issue is that writing software is quite different from writing hardware descriptions for synthesis. Softcore processors are low-cost yet efficient means of achieving hardware acceleration that are easy to program.

One of the most popular alternatives for hardware accelerators these days is the graphics processing unit (GPU). Today, they are programmed by explicitly parallel programming languages that ease the task of developers trying to use the GPUs efficiently. The use of GPU programming languages for the FPGA class of hardware accelerators has also been proposed [1]. The idea is that this will provide a unified language that is agnostic about the underlying hardware acceleration platform. To this end, CUDA has been used as a hardware description language for FPGAs [2]. However, using CUDA as a hardware description language would require developers to provide a significant amount of annotations in order that synthesis can be performed.

In this paper, we take a different approach to the problem. Instead of targeting CUDA (or CUDA-like) programs for structural instantiation of random logic, we propose a soft processor core that supports the CUDA model of parallelism. The soft core is easily customizable, and programmers need not have to struggle with descriptions for hardware synthesis. In this paper, we will describe in detail the realization of such a soft core, and present results of it performance.

978-1-4673-2845-6/12/$31.00 © 2012 IEEE

Our GPU-like soft core called *Guppy* (GPU-like cUstomizable Parallel Processor prototYpe) is based on an existing general purpose parameterizable soft core, namely the LEON3. There are many ways of producing a MPSoC based on such single cores. Guppy is novel in that the basic softcores are modified to execute CUDA-like threads in a lock-step manner to emulate the CUDA execution model. The execution and memory model in CUDA is unique, and reduces the need to rely on locking or message passing.

It would be quite a stretch to expect Guppy to be able to compete with a full-fledged GPU. Rather the aim is to have a low-cost, energy-efficient accelerator that would execute CUDA code the developer may have at hand. This is not too unreasonable as code for many signal processing tasks or matrix operations, for example, are readily available in CUDA. Guppy is particularly well-suited to accelerate code running in embedded processors processing smaller (especially integer and bit) data sets.

The CUDA execution model will be described in the next section. We will then describe the LEON3 architecture, and how we constructed our GPU-like softcore using it. This is followed by experimental results, and a conclusion.

## II. The CUDA Execution Model

The CUDA programming model captures explicit parallelism, as well as implicit synchronization derived from the underlying target platform. On GPU devices, function units (also called CUDA cores by nVidia) are grouped together into streaming multiprocessors (SMs). The cores in an SM are divided into execution clusters. In the current generation, there can be up to 4 such clusters, each having 48 integer cores, 16 double precision units, 16 load-store units, and 16 special functional units. All the cores in a cluster will execute the same instruction from a warp. A warp is a collection of user threads. The execution model organizes user threads into a grid of thread blocks. All the threads in a thread block are implicitly synchronized (warps benefit from SIMD execution, while multiple warps are guaranteed to be concurrent), and data is exchanged through the same piece of shared memory. However, threads in one block have no access to the shared memory of another block. For threads to share data across blocks, they must rely on a large, but slow global memory. We

refer the interested reader to the vast array of CUDA literature for a more detailed exposition.

## III. The LEON3 Architecture

The LEON3 is a open source softcore [3]. It is a synthesisable VHDL model that executes the SPARC V8 instruction set on a 7-stage pipeline with hardware multiply and divide units. It also has a fully-pipelined IEEE-754 FPU and a memory management unit.

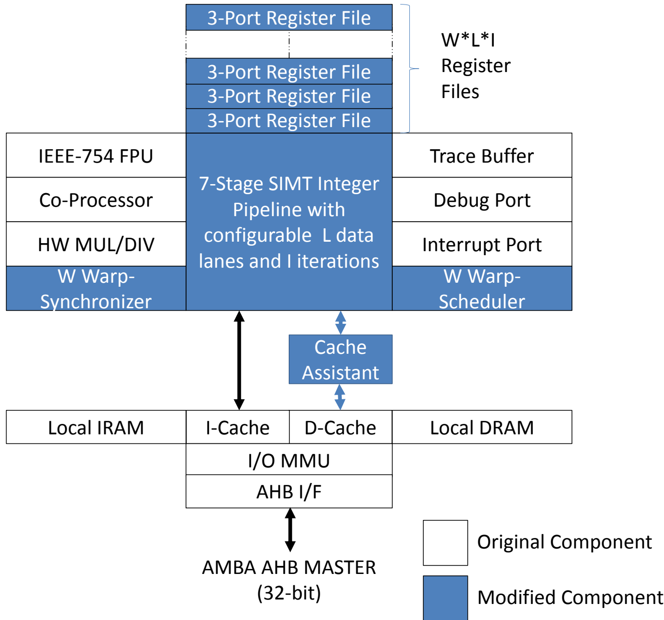


Fig. 1. The block diagram of Guppy.

## IV. The Proposed Guppy Architecture

Guppy is based on the LEON3 core. In order to support the CUDA concept of SIMD execution (warps), we replicated the data path to obtain the equivalent of multiple hardware threads. The following describes the modifications.

### A. Pipeline Parallelization (Parallel Data Lanes)

Guppy is built around a key modification to the original LEON3. The integer pipeline has been widened such that a fixed (user configurable) number of *data lanes* are executed in parallel. The block diagram of Guppy is shown in Figure 1. All parallel lanes (pipelines) have their own register files, and they execute the same instruction but with different data in a way much similar to Single Instruction Multiple Data streams (SIMD) paradigm, where the same instruction is applied to many data streams. Such implementation is particularly efficient for computations such as vector or matrix addition/ multiplication. Widening the data path imposes additional pressure on the data cache bandwidth as there are now multiple data streams per request (either from a load or a store instruction) instead of a single data stream for each request in the original pipeline. We handled this memory bottleneck through a cache assistant.

### B. Iterative Instruction Execution

In order to support as many threads as possible without undue explosion in the target FPGA resource requirement, we need to multiplex several threads onto the available lanes. To do this, we introduced additional logic that would re-execute the same instruction for threads whose thread ID is greater than the number of actual data lanes. For example, if we have 32 threads but only 8 data lanes, we will split them into 8 groups with 4 iterations each. This yields a 4-cycle SIMD core because now each lane must repeatedly execute the same instruction 4 times, each for a different thread. As with the number of data lanes, the number of iterations is also customizable. This modification further increases the size of the register files, as well as the length of the pipeline initiation interval. The number of register files would be equal to the number of data lanes multiplied by the number of the iterations.

### C. Cache Assistant

GPU caches need to offer a very high bandwidth as they must handle several requests (addresses) per clock. A multibank or multiported arrangement of memory arrays would be an architectural solution for this problem. As mentioned before, the LEON3 data cache supports a single data stream per request. We came up with a work-around solution for this problem by implementing a cache assistant unit that is situated between the wider Guppy pipeline and the original LEON3 data cache. The function of this unit is to fetch and serialize the parallel data stream for each request from the Guppy pipelines. Accordingly, it will communicate with the LEON3 data cache on a single-stream basis. Then it fetches and parallelizes the responses from LEON3 data cache before responding to the Guppy. In other words, parallel requests from Guppy are multiplexed onto the serial communication to the LEON3 cache, and responses are de-multiplexed accordingly.

The cache assistant consists of two parallel circuits. The Guppy Core-Cache Assistant (AC-CA) circuit handles of communicating with Guppy pipelines whereas the other one, the Cache Assistant- Data Cache (CA-DC) circuit, deals with the data cache. Both circuits exchange information of interest via a scoreboard which gets updated by both circuits in a conflict-free manner listing the addresses along with their associated data chunks to be stored/ loaded.

The inclusion of the cache assistant introduces an additional latency to memory transactions between the data cache and the processor pipeline. This allows the processor to continue executing other instructions when one data stream is waiting for a load or store instruction to be serviced. With the multiplexing support of the cache assistant, it is possible to deal with more data streams than the product of the data lanes number and number of iterations.

### D. Warp Scheduler

Memory latency along with the cache assistant overhead would make Guppy idle for a relatively large number of clock cycles. For instance, if we have 32 memory access instructions,

it would take at least $32 \times 4$ clock cycles for the memory transaction to be processed. Without any architectural solution, this would be a major obstacle in dealing with a larger number of data streams. To support more parallel instruction streams than there are actual physical lanes, a scheduler is needed to switch from one group of instructions (warp) to another. The warp scheduler collaborates with the cache assistant to schedule a warp that has already completed its memory transaction, and unschedule one that is about to engage in a new transaction. Currently, the selection mechanism for a specific warp among a group of ready ones is based on a priority encoder circuit that changes its priority levels over the clock cycles by a linear feedback shift register (LFSR) circuit.

### *E. Branch Predication*

For the Guppy to manage and execute hundreds of threads efficiently, it is necessary that it employs a single-instruction multiple-thread (SIMT) architecture in which one instruction can be applied to multiple independent threads that are free to diverge and branch. To support control divergence, we made use of the instruction annulment mechanism present in the SPARC V8 architecture [4], and implemented a branch predication circuit [5]. The annul signal through the data lanes is modified to selectively activate/deactivate the lanes according to their condition codes. The current implementation supports predication for `if` statements that are not followed by `else` conditions. However, this is not a major limitation as it is easy to translate `if-then-else` statements to `if-then`s.

### *F. Warp Synchronizer*

As warp switching is based on memory transactions, we must have an additional mechanism that enforces synchronization between multiple warp executions. In other words, we need a synchronization barrier. Currently, the barrier is implemented in hardware based on a specific address within the program counter (PC). In the future, we intend to use a software-based warp-synchronizing mechanism instead.

## V. Synthesis Results for Guppy

We have synthesized and tested a version of the Guppy with two warps, two iterations and four data lanes on a Xilinx Virtex 5 FPGA [6] achieving an operating frequency of 70 MHz. We used the LEON3 monitor and debug software, GRMON, along with Xilinx board ML510 [7] to verify the correctness of the implementation in executing an application that performs the multiplication of two matrices. Table I shows the resource utilization and running frequency of Guppy compared to the original LEON3.

| | **LEON3** | **Guppy** (2 warps, 2 iterations, 4 lanes) |
|---|---|---|
| Number of Slices | 10493 (51%) | 13877 (67%) |
| Number of Reg | 12717 (15%) | 17519 (20%) |
| - used as flip flops | 12716 | 17046 |
| - used as latches | 0 | 472 |
| Number of LUTs | 22036 (26%) | 30956 (37%) |
| Minimum Period | 12.455 ns | 14.256 ns |
| Maximum Frequency | 80.289 MHz | 70.146 MHz |

TABLE I
Synthesis results for Guppy on Virtex 5.

## VI. Guppy's Performance

In this section, we compare the performance of Guppy with that of the LEON3 using a matrix multiplication application. The LEON3 suffers from a single type of latency or penalty which is the data cache miss penalty whereas Guppy has the additional latency of warp switching and synchronization. It is important to note that the performance speed up is mainly affected by three factors:

1) Code optimization: LEON3 instructions do not take the same number of clock cycles. As the parallel code exhibits significant overhead due to index manipulation, code optimizations could result in better performance.
2) Data cache configuration: The original data cache deals with a single data stream for each request to the LEON3 that may be able to make use of the spatial and temporal locality in the data stream to improve its hit ratio. However, this is not the case for the Guppy. Consider a configuration of 4 warps, 2 iteration, and 2 lanes. When a warp has a pending request from the data cache, each warp expects 4 data streams with different spatial and temporal locality patterns to be provided by the data cache. This may yield worse performance compared to when there is only one warp. In other words, for the same Guppy data cache configuration, we expect a higher miss rate as we increase the number of warps, iterations, or lanes. Increasing the cache size and/or associativity could alleviate the sitation.

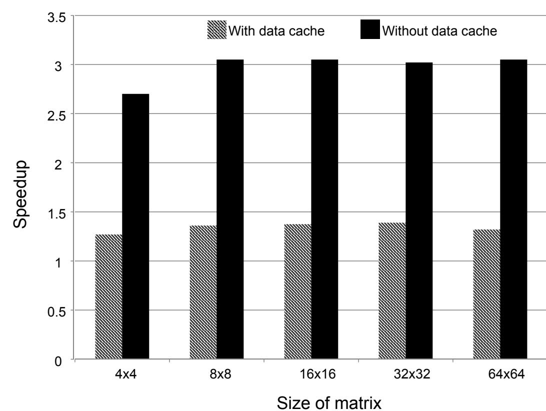


Fig. 2. Speedups of Guppy over LEON3 for different area sizes.

3) Guppy configuration: The number of pipelines (lanes), iterations, and warps affects the speed up significantly. By increasing the number of data lanes, we would expect a speed up that approaches the number of data lanes. By increasing the number of iterations, we will

still get a performance improvement due to the parallel lanes, but it would also increase the miss rate of the data cache. Having more warps increases the switching and synchronizing overhead which is detrimental to the overall performance.

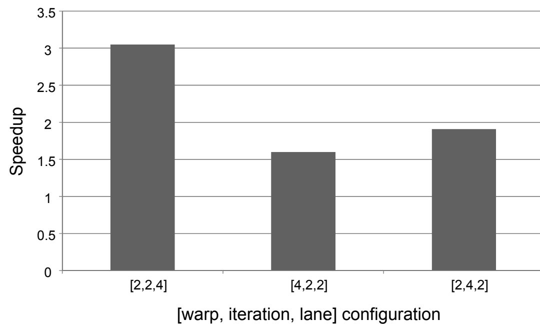


Fig. 3. Speedups of Guppy over LEON3 for different area Guppy configurations.

Figure 2 shows the performance improvement of the Guppy over the LEON3 for a matrix multiplication program using different array sizes. In this experiment, Guppy was configured with 2 warps, 2 iterations and 4 lanes. The speed up in this experiment was mainly limited by the data cache miss penalty and the warp switching and synchronizing overhead. The maximum speed up we obtained for this configuration was close to 1.4. To investigate the effects of Guppy's overhead on the speed up, we omitted the data cache miss penalty in second set of bars of Figure 2. As can be seen, the speedup is close to the ideal. It is obvious that data cache misses account for the most significant performance loss in Guppy.

The second component of Guppy's overhead is the warp switching and synchronizing overhead. To see the effect of this component, the number of warps, iterations and lanes was varied while keeping the total number of physical threads constant. Figure 3 shows the effect of varying the Guppy configuration on the speedup. In this figure, the data cache miss latency has also been omitted. The speed up depends significantly on the number of data lanes as it constitutes its upper bound. The number of iterations seems to have a negligible impact. However, changing the number of warps does impacts performance as the warp switching and synchronizing overhead increases with the introduction of new warps. In summary, the number of data lanes has a positive impact on performance whereas both the number of warps and the current data cache configurations have a negative impact.

## VII. Related Works

There are numerous "C-like" behavioural languages for programming FPGAs. These includes Handel-C [8], SystemC [9], and others [10]. It is generally accepted that such languages will ease the adoption of FPGAs. However, it is necessary to extend C with parallel constructs so as to take advantage of hardware parallelism.

The recognition that the GPU programming languages can form the basis for programming FPGAs has motivated a number of groups to use CUDA as a hardware description language. The most representative of these is FCUDA [2]. It is able to convert CUDA code into hardware on the FPGA. However, the main drawback is that it requires a substantial amount of annotations. Separately, there is a major effort at Altera to use the other major GPU programming language, namely OpenCL, for programming FPGAs [11].

The aim of Guppy is a lot more modest. It does not attempt to realize CUDA code directly as hardware. Rather, it is a softcore processor that supports CUDA code execution. The goal is to allow for the customization (either automatic or semi-automatic) of the softcore for the computation at hand.

## VIII. Conclusion

This paper introduces Guppy, a softcore processor built from an existing general purpose in-order LEON3 softcore processor. It was never a goal of Guppy to outperform GPUs especially on thread-parallel applications that needs to process a huge amount of floating point data. Instead, the aim is to have a softcore accelerator that can execute CUDA code on smaller (especially integer and bit) data sets in a cost-effective manner. Despite its significant compute capabilities, the cost and energy requirements of a full blown floating point GPU cannot be justified especially in embedded processing. However, our modest Guppy softcore readily outperforms the single LEON3 softcore, and is efficient even for small data sets. Its CUDA programming model also circumvents the problem of getting software developers to work on hardware description in order to accelerate their code. Our long-term goal is to run CUDA-like code on future versions of Guppy. For that we will need to work on a CUDA-based compile chain (which is now open source), optimize the architecture further, and perhaps extend it to floating point code. We also would like to complete the memory hierarchy support to include local shared memory.

## References


[1] A. Papakonstantinou, *et. al.* "High-performance CUDA kernel execution on FPGAs." Proc. of the 23rd International Conference on Supercomputing. 2009.

[2] A. Papakonstantinou, *et. al.* "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," Proc. of IEEE 7th Symp. on Application Specific Processors, pp.35-42, Jul 2009.

[3] Aeroflex Gaisler. LEON3 Processor. http://www.gaisler.com

[4] Oracle Corp., *The SPARC Architecture Manual Version 8.*

[5] S.A. Mahlke, *et. al.* "A comparison of full and partial predicated execution support for ILP processors," Proc. of 22nd Ann. Int. Symp. on Comp. Arch., pp.138-149, Jun 1995.

[6] Xilinx Inc., *Virtex 5 Family Overview*. Feb 2009.

[7] Xilinx Inc., *ML510 Embedded Development Platform User Guide*. June 2011.

[8] Celoxica Ltd. *Handel-C Language Reference Manual*. 2001.

[9] Accellera. *SystemC Synthesizable Subset Draft 1.3*. Aug 2009.

[10] Mentor Graphics. *Catapult C Synthesis Overview.* http://www.mentor.com/esl/catapult/overview.

[11] Altera Corp. *Implementing FPGA Design with the OpenCL Standard.* Nov 2011.