Highly Configurable Power Virus for GPGPUs

Mike Verlinden, Riccardo Gregorat, Nicholas Kelly

***Abstract*—As GPUs are becoming common-place co-processors, methods for improving power usage without affecting performance are being analyzed. Currently, GPUs are transistor-dense devices, with many independent cores, yielding high power usage. One of the concerns shared by both CPUs and GPUs is how much power they will use when operating at peak performance and how to develop programs that will allow developers to test their system when at or near that point. For multicore microprocessors, tools have been created for developing power viruses that can stress test how well the system handles the additional power and heat. These tools allow the developer to create tailored power viruses by providing parameters of the system to test without knowing the fine grain details of the every functional unit in the processor that would be necessary to create the same kind of program by hand. However, this kind of tool has not been created for GPGPUs and existing multiprocessor capabilities for general purpose microprocessors cannot be used for this purpose. We created a tool to create power viruses for various GPU architectures using GPGPU-Sim-Wattch to simulate the virus and measure its power usage. In the end we were able to create power viruses for the GTX 480 and Quadro FX5600 that came close to or exceeded the thermal limits for these cards and discovered correlations between certain hardware features and the corresponding parameters to our code generator that get close to producing the best power viruses. We also found Plackett and Burman design to be very helpful for tuning the search space for our genetic algorithm and for discovering trends in our results.**

I. Introduction

While GPUs were originally used for graphics intensive programs and rendering, they are increasingly used in other domains where its highly parallel nature and capabilities can improve performance. Currently, GPUs are transistor-dense devices, with many independent cores, yielding high power usage but the possibility for high throughput and performance. GPU manufacturers have added capabilities for utilizing GPUs in various purposes, but they still hide many of the underlying architectural features and interfaces in the GPU. This allows them to make frequent and radical changes to each GPU across generations that cannot be accomplished on CPUs; however, the rapid development cycle and significant changes between generations can make many of the test and verification aspects of the design harder. One of the important aspects of design and verification is determining appropriate thermal limits and power usage. It is not a simple matter of summing up the power usage across each part of the chip since it is generally impossible to have every part of the chip operating at peak power instantaneously. The typical way to find maximum power usage/thermal limits for a particular architecture is to create a program that uses as much power as possible, known as a power virus. The amount of power it uses is measured on the physical chip if available or using a simulator model. In either case, this is often done by hand and requires a significant amount of knowledge about the underlying system in order to create finely-tuned power virus. Additionally, it is difficult or even impossible to verify that the power virus created is actually representative of what the most demanding program would be. To help alleviate this burden, others have developed machine learning algorithms to create power viruses. These algorithms have the benefits of requiring less effort and domain-specific knowledge to create power viruses. Additionally, they can give a good indication that the maximum power of the virus has converged on a global maxima (given sufficient runtime).

Although these automated tools have proven helpful for creating power viruses for CPUs, they do not easily translate to GPUs due to their unique feature set and lack of CPU features. GPUs have different sets of memories and memory hierarchies that are shared across a large number of threads which run on multiple processors at once in small groupings of threads called warps (32 threads). GPUs also do not have as rich an instruction set as most CPUs but include many unique instructions that aid in computation. The fact that current solutions are insufficient lead us to take the insights and lessons learned in creating power viruses using machine learning algorithms for CPUs and apply them to creating a similar framework for GPUs. Additionally, since we do not have access to the GPUs to directly run our programs and measure results we will have to use simulation. Thus, we propose to use GPGPU-Sim to simulate different GPU architectures that we will create power viruses for. Finally, we will take the results from the GPUs available in GPGPU-Sim (with GPU Wattch) to construct an analytical model or other insights that would aid in creating power viruses for future architectures. The likely result would be a model that gives a starting point (the 90% solution), yielding narrower bounds on the search space and allowing the creation of a power virus much quicker and easier.

II. Related Work

Power viruses have proven very helpful for stress testing systems and determining thermal limits. Often power viruses are created by hand for a particular architecture; however, a power virus created for one architecture often is not useful in the next generation. Currently, some tools have been created for generating power viruses for single core processors. Ludvig Norinder described a method for breeding power viruses for ARM processors in his thesis [1]. He did a thorough job of describing the process for creating the code generator and genetic algorithm that would create configurations for the power. Similar to other code generators, he varies the instruction mix, memory usage, stride patterns, dependency distances, basic block size, and branch behavior. One of the helpful contributions we took from his work is the use of a

seed gene that will seed the random number generator (RNG) in the code generator. During generation, the seed allows the instruction and memory mixes for an individual configuration file to deterministic, while still allowing additional variation to be added later.

While most tools focus on creating viruses for single cores, MAMPO [2] generates optimal power virus workloads for multicore systems that consume between 40 and 89 percent more power than when a single core power virus is run on multiple computers. MAMPO uses a genetic algorithm that creates an abstract homogenous and/or heterogeneous multithreaded workload. It creates a set of threads that each run through a series of inner and outer loops with particular mixtures of instructions, memory operands, branches, dependency distances, and stride access patterns. In many respects, the work from MAMPO relates quite well, since a GPU can be thought of as a homogeneous multicore processor. However, there are numerous differences in the way that a GPU operates, and the special features it has that are not well captured by the MAMPO design. MAMPO can also model high-level program behaviors such as producer-consumer models, irregular sharing patterns and others. However, these do not seem to fit the GPGPU model well or would be likely to introduce branch divergence issues.

Similar to MAMPO, SYMPO [3] uses genetic algorithms to generate power viruses that maximize the power usage across the entire system, not just the CPU. The need for system level power viruses and stress tests is such that other parts of the system can contribute significantly to the power usage of the system, even though the CPU is the biggest contributor. One of the key components of SYMPO is taking into account DRAM power usage under different circumstances. The goal of the tool is to not necessarily produce the most power hungry viruses possible, but instead to create programs which mimic the actual power usage of a real program. One of the more interesting findings from SYMPO was that top-performing viruses heavily leverage DRAM power usage. Thus, showing that a power virus which only maximizes CPU power is unlikely to be the best system-level virus.

SYMPO also implemented their genetic algorithm using SNAP from IBM which used some unique characteristics and techniques for exploring the search space. SNAP runs until a specified fitness level is reached as opposed to being open ended or reaching a certain number of generations. The initial population is randomly generated given a set of bounds for each parameter (which needed to be carefully chosen in order to not overly constrain the search space). Rates for mutations, elitism, and crossover can be established and appear to work at the individual, not gene level (which only matters for mutations which are not necessarily applied to every new individual produced). The selected rates for mutation, elitism, and crossover are important for helping the search converge quicker and for insuring that values are not chosen which prevent convergence.

While we took some inspiration and lessons learned from these systems, GPUs have special characteristics which do not relate well to traditional CPUs. GPUs have shared registers, scratchpad memory, and texture memory in addition to global memory which makes the job of creating a memory model to effectively use them harder but much more important consideration. The two loop method described in most implementations for setting up stride access patterns and reusing code does not make as much sense for a GPU since most of the reuse in a GPU comes in the form of a large number of threads instead of large loops. Often, each thread will work on its own section of memory with varying amounts of overlap between threads depending on the application. Additionally, while basic blocks, branches, and synchronization are all possible in GPUs, they are known to cause branch divergence. Effectively, this prevents the processor from realizing maximum throughput and presumably maximum power usage. Overall, the structure of our system will be similar but heavily tailored to the strengths and limitations of the GPU, as opposed to the more generalized framework that the others have created.

III. Background

Since we do not have access to physical GPUs, we will be using GPGPU-Sim for simulating the generated kernel. GPGPU-Sim uses GPU-Wattch to calculate power usage based off of the simulation results. GPU-Wattch uses McPAT as its underlying power model, so understanding and utilizing these three tools and their interaction is key to understanding our results.

A. GPGPU-Sim

Ali Bakhoda, et al. [4] discuss in great detail their GPGPU simulator (GPGPU-Sim) which implements compatibility with NVidia's CUDA programming model. The compilation and execution flow of CUDA for GPGPU-Sim is made to mirror that of CUDA on real hardware. The CUDA program is first transformed into C code for both the CPU and GPU. The GPU C code is then compiled into the native ISA for the GPU via the PTX compiler, which is subsequently assembled by ptxas. Next, all the code is compiled into a single executable in machine code (SASS) for GPGPU-Sim. Additionally, the code is not assembled by ptxas for the simulator, but is used as-is to be run by the PTX functional simulator.

Subsequently, the authors benchmark a chosen architecture with a parallelized AES encryption algorithm; Breadth first search on a graph with over 65,000 nodes; a 3D Laplace solver; a weather prediction program; and ray tracing on a 256x256 image as well as other interesting benchmarks. These programs function by running a parallelized compute kernel containing a grid of scalar threads. This grid is subdivided into cooperative thread arrays (CTAs) which are assigned to one of several shader cores on the GPU. These CTAs become converted into warps which are groups of threads waiting to execute the same instruction over multiple data sets (SIMD). The scheduler chooses a warp to run among those that are ready and issues a portion of the warp into the pipeline based on the width of the processor. The pipeline configured in GPGPU-Sim by the authors of this paper for each of the 32 shader cores is 6 deep and 8 wide, meaning that a 32-thread warp takes 4 clock cycles to issue.

Some of the important performance issues the authors discuss come from the communication and synchronization of various CTAs being executed on the multiple shader units. One issue that occurred with the breadth first search benchmark they tested was referred to as CTA load imbalance, which occurs when CTAs awaiting assignment to a core might experience sub-optimal assignment. This can occur when multiple CTAs are waiting, and multiple CTAs may be assigned to a specific shader unit. If one shader core finishes slightly earlier than another, a net performance degradation occurs since each CTA must run to completion on the resource to which it is assigned. Performance would be expected to improve if these CTAs did not have to be assigned simultaneously to the same unit.

After tweaking other various parameters of the simulator to have 28 shader cores 28, a warp size of 32, and 16384 registers per core (for the full list refer to the article) they obtained experimental results with the simulator indicating a maximum IPC of 224 (which comes from the shader count times the core issue width). The instruction mixes for each of the benchmarks used primarily consist of ALU operations. Some benchmarks have a high control-flow instruction count; however, these do not directly translate to high branch divergence if most of these branches agree with each other, as is the case for the N-Queens solver benchmark. Low warp occupancy typically translates to high branch divergence. However, for the set of benchmarks tested, they found that the NN (Neural Network) benchmark had the lowest warp occupancy, though this was not due to branch divergence since it had low control flow instruction count. Instead, it was due to the fact that each kernel had only one or two threads active, taking up most of the execution time.

The authors tabulated access percentages to the various memory types. However, very little commonality can be gleaned from this table aside from the fact that all benchmarks use different types of memory available to the cores to different degrees with half of the benchmarks concentrating on memory accesses to texture and constant memory. They also note that more threads do not always translate to optimal performance. It is found that in some cases, such as the AES encryption algorithm, where the performance decreased as the number of CTAs per core increased. This translates to memory contention as more concurrent CTAs are requesting memory. Another important conclusion they drew is that the interconnect bandwidth affects performance much more than latency of the routing units themselves.

While the simulator is essential to our work, some of their findings have proven helpful as well to our power virus study. With the variable configurations available of GPGPU-Sim (such as the number of cores or CTAs available to each core) the power virus will likely vary between architectures. It is possible that ideal parameters can be tailored to specific GPUs with different texture/constant cache sizes and register file sizes though our results for the two GPUs we tested generally did not use either in their power viruses. Additionally, memory accesses and thread counts were found to contribute significantly in our results likely relates back to their analysis from AES. While our code and results do not always show an easy correlation to their findings we certainly kept them in mind as we progressed.

B. GPU-Wattch

Performance monitoring has traditionally been the main concern of users of GPGPU-Sim. However, Jingwen Leng, et al. [5] discuss a particular modification named GPU-Wattch which provides valuable power profile data as an enhancement to GPGPU-Sim. The authors studied and implemented a power model based on the various active components of the NVidia-Fermi GPU architecture. However, they attempted to abstract as many of the individual components of the architecture in order to assure the configurability maintained by GPGPU-Sim. Many of the aspects incorporated into their power model were done indirectly via the timing and power modeling tool known as CACTI [6]. The first part of the resulting power model is the sum of all max dynamic execution power, multiplied by an activity factor in order to abstractly represent the fluctuations in power usage of the individual component. The second part of the power model consists of adding the power consumed by idle shader units and the overall leakage experienced by the GPU.

Unlike typical CPUs, the register files for GPUs are distributed and their structure depends on their size and the bandwidth between shader units as well as the register file blocks themselves. Also, since the register files are distributed, a crossbar interconnection system is required to transfer data between the register blocks and operand collectors. Since CACTI is accurate for modelling interconnects, it is utilized to model a 16-bank crossbar system for the register file. The shared memory also utilizes a crossbar which is implemented as the Fermi version of 32 inputs and outputs (32x32). Functional units on the other hand were modeled using the Synopsys power profiler to estimate power consumption of the designs based on 45nm technology while including an annotation for switching frequency. One of the larger pieces of the model is main memory. Main memory must be modeled, but they do not have direct access to the GDDR power modeling information. So, they generate a custom power formula that depends on the number of memory access operations scaled by the amount of energy they consume. All of this is normalized by the execution time for the operations themselves.

They also described their results from running various micro-benchmarks for the purpose of testing their model. Since typical benchmarks are overly broad and may at times refrain from testing all parts of an architecture, micro-benchmarks are used to hone in on a specific area for testing. This was done to test and validate the resulting power model and apply stressors to various aspects of a microarchitecture to ensure better coverage. In order to refine the power model that they have initially generated, they implemented a feedback process between performance counters and the dynamic power model. They looked at a few major design factors such as individual component stress, which is excellent for targeting potential modelling inaccuracies of components such as functional units, and access patterns. This effectively indicated the diverse ways in which dynamic power might result in inaccuracies in timing assumptions.

These micro-benchmarks are then run on two different architectures, the GTX 480 and Quadro FX5600. This was done on the real hardware as well as on GPGPU-Sim for comparison purposes, where the resulting error was 9.9% and 13.4% respectively. Furthermore, important modifications that scale down power usage in GPUs are also analyzed in this paper. Coarse-grained and fine-grained dynamic voltage and frequency scaling (DVFS) are utilized which scale down the 45nm device's voltage and frequency after larger and smaller observation window of chip activity, respectively. With roughly a 3% performance loss, Coarse-grained DVFS achieves 13.2% in energy savings while fine-grained achieves 13.2% in energy savings. Also, branch divergence is determined to cause noticeable performance degradation for GPUs, but was exploited in this paper for clock-gating individual lanes. The resulting power savings across the micro-benchmarks used were found to have a geometric mean of 11.2%.

The accuracy of our results is ultimately limited by the performance of GPUWattch. Fortunately, they also showed that their trends for their various power measurements matched the actual trends in hardware. Relative accuracy can be more important for our GA than overall accuracy since that means that the GA is not exploring erroneous paths (though overall accuracy is still important). Additionally, the distribution of power usage they found when running GPU-Wattch on the various micro-benchmarks was found to have three major contributing components: DRAM, execution units, and register files. Therefore, these three components deserve special attention when looking at the breakouts of power usage available from GPU-Wattch. Finally, they also mention that there are certain graphics specific units that are not included in their power model. These units are generally hard to use or unusable by GPGPU code, but could still limit our power viruses effectiveness.

C. McPat

McPAT [7] is another multi-core power simulator that has been adapted to model power consumption of GPUs and is the underlying model used in GPU-Wattch for modeling power along with CACTI. GPGPU-Sim power modeling was implemented using McPAT concurrently with GPU-Wattch. McPAT models a multi-core processor with shared caches, memory, and I/O that are subdivided into additional components which can be further divided into subcomponents that can be modeled separately. Similar types of the lowest level subcomponents can be modeled with the same type of circuit-level models that can be configured to represent the various types of components. For extending McPAT to GPUs, they created a similar hierarchical model of components with changes to reflect some specifics of a GPU core and some of the special shared structures like the shared registers, memory scratchpad, and texture memory. They also include performance counters on each of the components in the GPU. Each access to a component will expend a certain amount of energy depending on its type, though not at the granularity of bit transitions. It was unclear if this means that simple multiplications and divisions are not handled differently or if it just means that specific complex bit patterns will be treated the same way (which if trying to determine max power consumption, limits the possible precision with which we could generate our inputs). They modeled each of the streaming multiprocessors with a set of three types of functional units: a streaming processor for normal arithmetic functions, a load store unit and a special processing unit for complex instructions (sine, sqrt, etc.) as well as the models for the shared memory and registers. Simulation and power measurement use MacSim for architectural and timing simulation as well as DRAMSIM2 for memory simulation and power measurement in addition to the rest of McPAT.

To calibrate and validate their model, they measured the power usage of a NVidia GTX 580 on a series of benchmarks as well as in an idle state to get baseline estimates of how much power is used when the functional units are active. They used McPAT to create power models for the parts of the functional units they could find sources or documentation describing them. For parts that are not specified, they set the values by finding which ones gave the least error when running the various parts of the benchmarks knowing what the mix of active units should be. They used the same process for determining the power usage of the L1 and L2 caches but also had the assistance of some better levels of documentation (though not sufficient to use directly in McPAT). They also had to model the special shared memory structures as well as the leakage and clock distribution power usage. For all of these estimations, the original physical measurements were used as training data to select the values that would provide the smallest error after using additional techniques to try to isolate the contribution of that component to total power.

In the end they tested 16 different benchmarks which had a geometric mean error of about 13% and follow the trends in power over time closely. It would be interesting to see the difference between the best viruses that were created with each model, that is likely beyond this project. One of the more interesting points of analysis that relates to what we will be working on was their breakdown of what used up the most power. The streaming processor, memory, and clock were the biggest contributors to power usage (though it may be partially due to the types of benchmarks they ran combined with power gating). Maximizing the memory usage and streaming processor will be key to achieving the highest power usage.

IV. Methods

A. Overview

Our power virus generator consists of three parts: our GA implemented in DEAP [8], a code generator that will produce the power viruses, and GPGPU-Sim and GPU-Wattch which will simulate the program and model the power. First, the GA creates a configuration file for the code generator. Then, the code generator creates a CUDA C program that we compile with NVCC with no additional flags. The resulting program is run using GPGPU-Sim which provides the information needed for the GPU-Wattch to produce the average and maximum

power figures for the kernel. We use the average power as the fitness parameter in the GA which will repeat the evolutionary process until the user terminates it. Figure 1 shows a diagram of the full system.

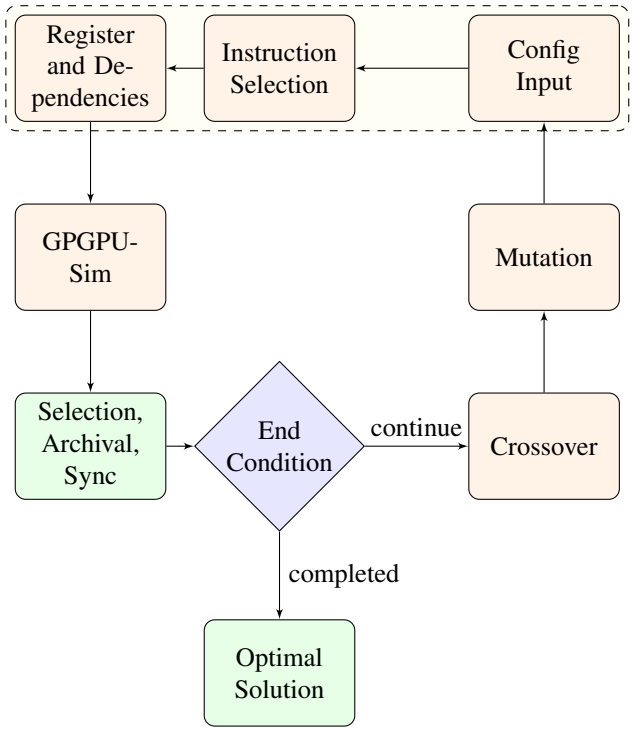

Fig. 1: GA and Code Generator Flow

B. Challenges & Difficulties

The primary difficulty we faced was the simulation time of GPGPU-Sim. It requires the program to run for at least 500us of simulated time in order to get an accurate power measurement; however, doing so can take over twenty minutes on our best computer running a single instance. We had each machine run as many instances as it had cores to support them, which increased throughput but each run can take 40 minutes or more to complete. We had six machines working on the algorithm at various points. Our fastest machine was able to simulate 1200 cycles a second (with 8 logical cores) and our slowest only 200 (two cores) for a single instance. We found that multiple instances can run at once since it is single-threaded but each instance can take about twice as long when running more than one at once (it does not scale linearly so 8 threads is not significantly worse for a single instance than 2). Configuring the GA to create programs that would run long enough to get good results and explore the search space without wasting simulation time was key to ensuring that we finish in a reasonable amount of time. We did more work on the Quadro run to effectively bound the search space ahead of time (described later). Finally, we had to design a method for running our GA so that it could allow multiple computers of varying quality to contribute to the solution without being a bottleneck. Additionally, many of the computers working on the solution were personal laptops that were used for other work and would not be working on the solution for large periods of time.

The second biggest difficulty we ran into was that GPGPU-Sim only has the four GPU models and only two of the models support GPU-Wattch. This will limit how many real world GPUs we can try to gather power viruses against for generating our analytical model. We considered varying the parameters directly in GPGPU-Sim, but this would not be mappable to a real GPU and thus might not have accurate power measurements. Ideally we would have many GPU models to tests against and be able to use most for training and generating the model and the remainder for validation. For now, we are focusing on developing the model for determining good starting bounds for the search space supplemented with results from Plackett and Burman analysis to setup narrow ranges on the search space to explore.

C. Plackett and Burman Design Usage

One of the more difficult parts of using a genetic algorithm is to create a large enough search space that it can find the right solution but not so large that time is wasted in local maxima. Since we expected simulation time to be the primary bottleneck, we tried to limit the search space using Plackett and Burman (PB) analysis. The goal was to identify which parameters mattered the most and set acceptable bounds on those and to determine if there are any parameters that are heavily biased in one direction that we should set accordingly. For our GTX480 run, we only used the results from our PB design to tune the code generator and identify which parameters we should set to zero. For the Quadro FX5600 run, we also used multiple instances of a PB design on the key parameters to establish good ranges for the search space. We ran it once to identify the most important parameters then we adjusted the range of values to search depending on which direction it favored. We then reran the PB design on the reduced set of parameters and evaluated the range each time. For the number of threads and some of the other parameters we found a value that when set as either the lower or upper bound was the direction the results always pointed to. This helped us get good starting bounds for the most important parameters by setting the min and max of the search space around the middle value. Below are some of the other findings:

- The number of threads was always the most important parameter. For GTX, the analysis showed it needed a higher number of threads than the Quadro.
- A lesser SFU weight was important on both runs so we set it to avoid using it or perhaps sparingly. The likely reason is that on some architectures only a fraction of the threads in a warp can use the SFU at a time limiting the amount of power it can use which we believe was the case for these processors (which have a ratio of 1 to 8 for their SFUs to threads in a warp).
- Float was much higher in weight than integer and integer was valued negative. It seems that floating point operations take significantly more power and there does not appear to be any value in including a mix of the two.
- Add and divide were the highest prioritized instruction on both runs but the GTX favored multiply more than simple

ALU instructions whereas the Quadro favored simple ALU instructions and did not generally use multiply.
- Shared memory was the most heavily used memory operand on both processors followed by Global/Parameter.

We also used our run with GTX to verify that the information we got from the PB design was useful. The early highest performing kernels each showed properties close to what the PB design recommended. Those kernels only used floating point instructions and never used the SFU or type conversions while showing similar memory usage patterns. They also used a high number of threads (when compared to the range the GA was searching) and transferred over fairly large amounts of data.

D. Genetic Algorithm

In order to explore and evaluate the narrowed (by PB design) search space, we used a customized Genetic Algorithm (GA). Our GA used the DEAP (Distributed Evolutionary Algorithms in Python) library [8], which allows for the creation of distributed GA's. Additionally, since GPGPU-Sim contains the most overhead (in terms of execution time) in our system, a lower-level GA implementation is not needed. Since we are only maximizing one variable, simple GA algorithms can be used. Essentially, a GA creates individuals with certain (initially random) genomes. After each generation, individuals are selected to be bred (crossover) and mutated to yield new individuals and the cycle continues. The genes that make up the genome correspond to the attributes of the solution. In particular, our genome consists of a python dictionary which is filled from a YAML configuration file which specifies the valid ranges of each variable in the code generator.

Overall, the GA process can be broken down into distinct sections: Initial Population, Evaluation, Selection, Mutation, Crossover, and Archiving. Our Initial Population consists of a grouping of forty randomly generated individuals within the ranges specified in a GA configuration file. The GA configuration file contains range and stepping information for each gene for the Code generator, which is finely tuned by the PB design. Evaluation consists of a multi-step process, beginning with generating a configuration file for the Code generator. Next, the Code generator is run to yield a synthetic CUDA C program. Finally, the program is compiled with NVCC and run with GPGPU-Sim. After GPGPU-Sim finishes, the total average power for all kernels is extracted and used as the fitness value for the individual. In the Selection phase, all individuals are run in a local (3 individual) tournament to select fit individuals. Additionally, the top five individuals are retained, no matter the tournament outcome. Mutation and Crossover are done after Selection to create a new grouping of individuals (offspring). In terms of mutation, we perform multiple mutations of random genes at a high mutation rate of 40% (due to duplicate individuals being created). Similarly, we use uniform-crossover at high rate of 30%. The reasoning behind high mutation and crossover rates was due to the initial findings and limited runtime. In initial runs, low (traditional) rates of around 1% provided many duplicates (individuals already evaluated). Thus, in order to speed up convergence (at a coarse granularity), avoid local maxima, and avoid repeated individuals, high mutation and crossover rates were selected (see the Future work section for more information). In addition, if the GA could not generate enough invalid offspring over a period of 10 tries, it would create a new population, only keeping the top five individuals (in order to increase diversity). The last stage consists of archival, where each individual and their corresponding fitness are saved into a file (hash table) for later retrieval and to detect individuals that had already been evaluated.

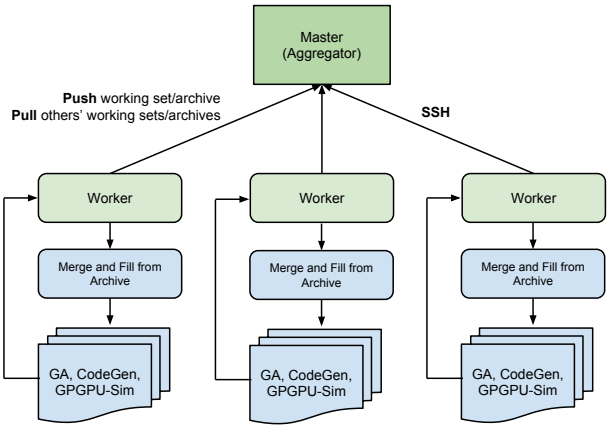

Fig. 2: Distributed GA

Due to the highly variable simulation time (ranging from several minutes to worst case one day), more machines were needed to evaluate as many generations as possible. Thus, we have created a distributed GA (see Figure 2) allowing for greater parallelism throughout the GA execution. The basic architecture of the distributed system is a main Aggregator with many Worker nodes. The aggregator simply serves as a central area to store files (server). Each worker first pulls each hosts' working set and archives from the server. Then their current (or initial) population is merged with each hosts' current working set. These individuals then go through a selection phase, are mutated and bred to create a set of new invalid individuals. Data from the various archives is used to complete (fill in fitness) individuals which have already been ran (based upon a unique hash for each individual). This mutation and fill-in process loops until enough invalid/new individuals are created (enough for number of GPGPU-Sim processes). During evaluation, working directories are created for each individual. In this way, multiple individuals can be evaluated at the same time (multi-threaded). Each working directory contains configuration files for GPGPU-Sim and a Makefile generated CUDA source. Each unique run is tracked as a hash (SHA128) of the code generator variables. Using the hash, each unique individual is archived and synced across other machines running the GA as well (by uploading data

to the Aggregator). Additionally, each worker is resilient to failure or termination. Since each computer is a personal computer, one might need to terminate the GA and work on another task. However, when the GA is resumed, it will start right back where it left off.

E. Code Generator

Once we generate the configuration file using the GA, we create an executable to run in GPGPU-Sim using our code generator. The code generator creates a CUDA C file (.cu) that includes both the CUDA kernel that runs in GPGPU-Sim and C code that initializes the GPU and transfers data over from the CPU. We compile using NVidia's compiler NVCC along with a few libraries from their SDK for inlining CUDA kernels in the C code. The CUDA kernels it creates have initialization code followed by a loop of configurable size, access pattern, instruction mix, memory operand mix, and iterations. The C code that launches the kernel follows the following steps:

1) Initializes Input/Output arrays on the GPU and CPU
2) Creates and initializes a Texture on the GPU
3) Creates a set of streams on the CPU that were each stream passes over its data then performs its computation
4) Wait for the output and frees the memory

The number of threads, threads in a block and number of streams to use for the kernel are configurable and set in the C code. The size of the input/output arrays is configurable but the texture size is fixed to fit in the smallest texture memory of the devices we worked with. Key to getting the highest performance possible is using streams to overlap memory transfers and execution. Normally transferring memory and launching a kernel are serialized by the CPU; however, CUDA includes functions for allowing the CPU to issue request asynchronously. Asynchronous requests have to be associated with a particular stream such that requests in a stream are serial but different streams execute in parallel. As an example, a set of four streams would display the following pattern of execution similar to pipelined execution: the first stream would transfer its data over, the second stream would begin transferring its data while the first stream begins execution, the second stream's execution would begin to overlap with the end of the first streams execution (presumably when the first stream does not have enough threads available to fill each processor), the third and fourth streams would repeat the same pattern until the fourth stream is running by itself and eventually ends.

Table I shows a breakdown of what parameters are settable and where they are actually used. Each item in the kernel has a weight representing the percentage of its usage and dependency distance which will ensure the desired spacing between uses and overrides the weight if there is a conflict (for example, high weight but high minimum dependency distances will result in infrequent usage). However, we did not use all the options available. Our PB design showed that floating point operations consume much more power than integer to the point that we should not use integer operations except for address calculations. This also means that we did not utilize conversions since everything would always be floating point. We chose not to use synchronization barriers and branches (except to prevent divide by 0 or negative square root faults) since both are expected to reduce the number of active lanes in a processor. Simple ALU instructions include AND, OR, shift, min, max, negation, and more. Not all of these are available for floating point instructions. SFU instructions operate on floating point numbers and include cosine, sine, square root, and inversion. The code generator places instructions and memory operands based off of the rules established in the configuration file and uses the RNG for breaking ties when more than one instruction/operand could be used. To keep the code generator deterministic, we include a seed value in the config file that seeds the RNG. Thus, changing only the seed will create different but similar kernels with the same distribution of instructions and operands (the C code will remain unchanged though).

TABLE I: Code Generation Parameters

Kernel Instruction Mix	**Kernel Mem Usage**	**CPU**
Simple ALU	Register Usage	# Threads
Add/Sub	Global/Param	Threads Per Block
Multiply	Shared	# Streams
Divide	Texture	Input Size
SFU operations	Constant	Output Size
FP/Int %	Local	
Conversion		
Conditional Branches		
Synchronization Barriers		

When running NVCC we use the defaults which will optimize to roughly the equivalent of O2 for GCC but does so for both the PTX intermediate code and the internal SASS. We originally wanted to have the option to generate both CUDA and PTX code, but we found out early on that not running the optimizers significantly lowers the power usage. Any PTX code we generate would be largely equivalent to its corresponding CUDA code and would end up being optimized the same way. One would have to turn off the optimizers on the PTX code to see significant differences but that would not be helpful in the long run. The only advantages using PTX might offer are to have finer control over the register usage and being able to use special instructions like fused multiply add (FMA). In the end, the optimizer was proficient in using the registers for power usage so the only question would be whether FMA or similar instructions would increase power usage.

V. Results

A. Comparison to Benchmarks

We began our analysis by running as many benchmarks as possible from the Rodinia GPU benchmark Suite and the various test programs from the NVidia SDK Toolkit. None of the benchmarks used as much power as some of our early power viruses. However, they were still useful for having a starting reference and for analyzing component power levels thanks to GPU-Wattch's power breakouts. Additionally, we attempted to run CUDALucas which is the GPU equivalent of MPrime; however, GPGPU-Sim did not support a recent enough version of CUDA to run the libraries needed for

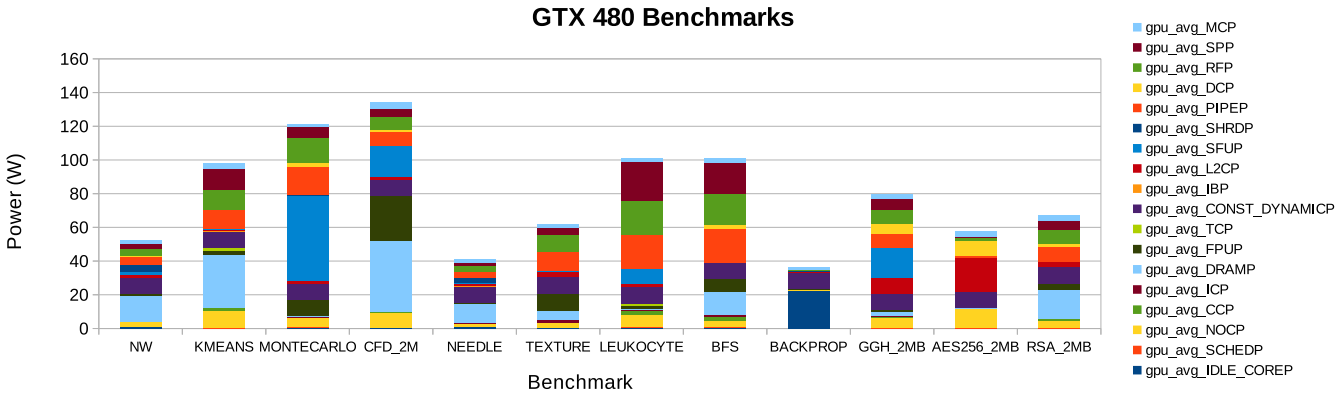

Fig. 3: Benchmarks Run on GTX 480

this kernel. These results could have been potentially useful to compare results gathered in SYMPO to our own. Using collected benchmark data, we were able to estimate how much power a particular component could use and make predictions about how much more additional power a particular power virus might use if it better utilized a particular unit. We also verified which components do not have high power utilization compared to other alternatives (i.e. SFU vs ALU). Figure 3 shows a breakdown of each of the benchmarks we ran compared to our top performing power viruses.

B. GTX 480

We ran our GA for the GTX 480 for 20 days before we had to end it early to start the Quadro FX5600 run. We had started the GTX 480 run before that, but had to make changes to the GA and Code Generator based off the initial results. The thermal limit for the GTX 480 is 250 Watts and the best result we found had an average power of 216 Watts and a maximum power of 241 Watts. We did not run the GA long enough to see it converge which means it may have achieved significantly better results with more time (+10-20 Watts), but the parameters to the GA may have prevented it from converging in the first place. It also may not be possible to reach or surpass the thermal limit for the GTX 480 in GPGPU-Sim since it does not model the shader specific units (polymorph engine, render output units, etc) which are included in each SM and whose contribution to total power is unknown. Figure 4 shows the improvement in power usage overtime. The highest seen average power stays steady for long periods of time then jumps quickly and is likely due to the way the GA is configured to make large, coarse changes between individuals in each generation. However, coarse jumps could indicate finding other local maxima, meaning exploration is still developing (not near convergence).

Figure 5 shows the per component breakdown of the power usage for the best performing power virus seen at each point. One of the trends we see is that improvements in power usage in one component usually cause a short term decrease in the power usage of other components. In the beginning, floating point power grows quickly. Afterwards, DRAM power increases quickly at the expense of floating point power to produce a marginal gain in overall power usage. Eventually floating point power rises again at the expense of DRAM power which never quite recovers to its previous high. It seems likely that if the GA was allowed to run long enough we would see DRAM power grow closer to it previous high with a temporary downturn in floating point power. We think that this is at least partially caused by the parameters we used in the GA that cause the large variation between individuals.

C. Quadro FX5600

We ran our GA for the Quadro FX5600 for 5 days. It looks like it was nearing convergence and had already come close to the thermal limit (171 W) for average power (165 W) and passed it for maximum power (175 W). We were able to get good results faster partially because it did not take as long to run each simulation but mostly because we used information from our PB design to set better bounds on the GA before it ran. Within the first couple of generations we were able to see a significant improvement in the power usage. Figure 6 shows the highest average and maximum power seen by the GA as it runs. Within the first day we produced results that were within 5 Watts of the best individual seen during the entire run. We also saw more steady improvement in both the average and maximum power as opposed to the large jumps seen in the GTX run. This is likely attributable to the design space pruning that we achieved with PB design that limited the impact of the parameters of the GA that cause large, coarse changes.

Figure 7 shows the per component power usage of the best seen power virus at that point. Similar to the GTX 480 improvement over time we saw that increasing power usage in one component might improve overall power but usually lowers power in another component. At first we see the floating point power grow until it reaches a local maximum which reduces the shared memory and DRAM contributions. Next we see DRAM grow to its original usage level which in turn temporarily lowers the floating point power component. In the end both appear to reach their maximums, at least as far as we have seen in this run, but the shared memory power does

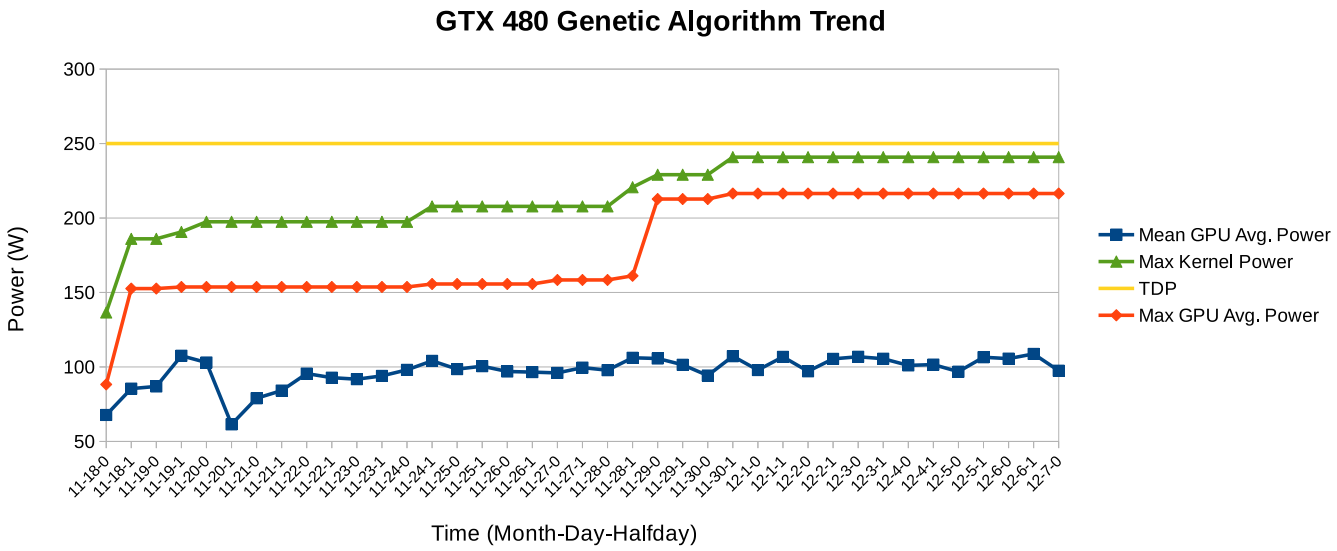

Fig. 4: GTX 480 - GA Timeline

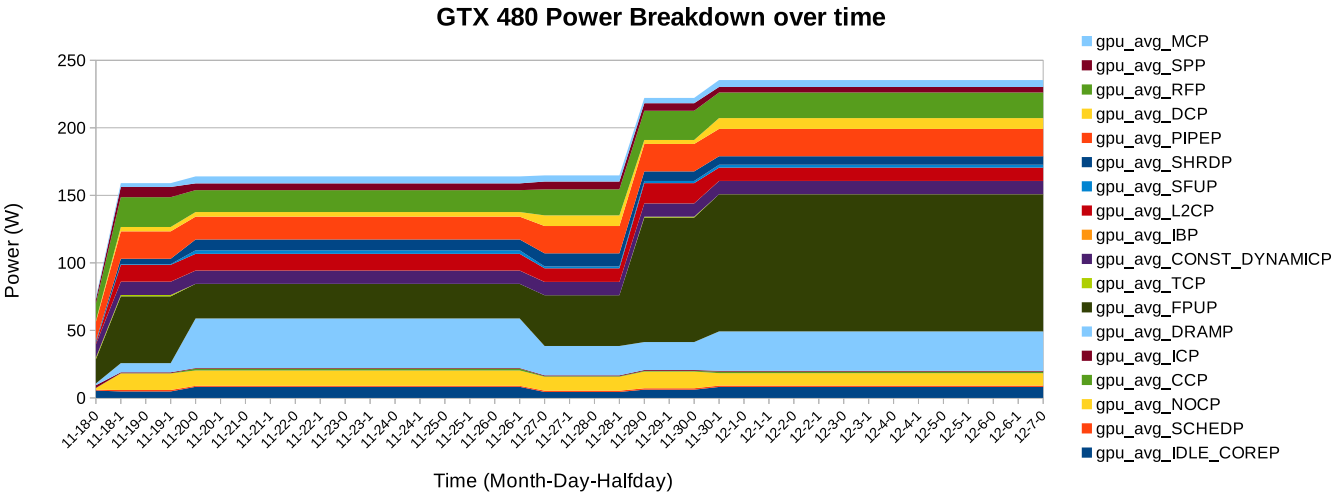

Fig. 5: GTX 480 - GA Timeline Components

not climb to its previously seen highest value. This might mean that shared memory power can grow by up to 10 Watts before we start to reach a point of marginal improvements. Even if shared memory cannot improve, the fact that there is idle power leads to the possibility of another component increasing.

D. Variation across seeds

As mentioned above, each configuration file has a seed for the code generator RNG to use when creating individuals. We left the value fixed for both of the GA runs because the values do not have a useful meaning as a gene. Since values that improve a results only work for that particular configuration, crossing over or mutating the value makes no difference compared to just selecting a random value (or in this case using the same random value). We also expected that varying the seed would not show too much variation between configurations with all the same parameters; however, when we tried varying the seed for our best performing power virus and generated 9 new power viruses, we found a significant amount of variation between individuals. Figure 9 shows up to a 25% difference between the best and worst individuals produced by varying the seed. We were able to see a noticeable improvement (162 W to 165 W); however, we saw much higher variation for results below the original value. Floating point and DRAM power seem to vary the most between seeds which is likely the result of getting a slightly skewed instruction or operand mix. As an example if the Global memory usage (which corresponds closely with DRAM power usage) is low then there could be large differences caused by the RNG where some versions use 2, 3, or 4 accesses, thus a small difference skews the results. Alternatively changes could

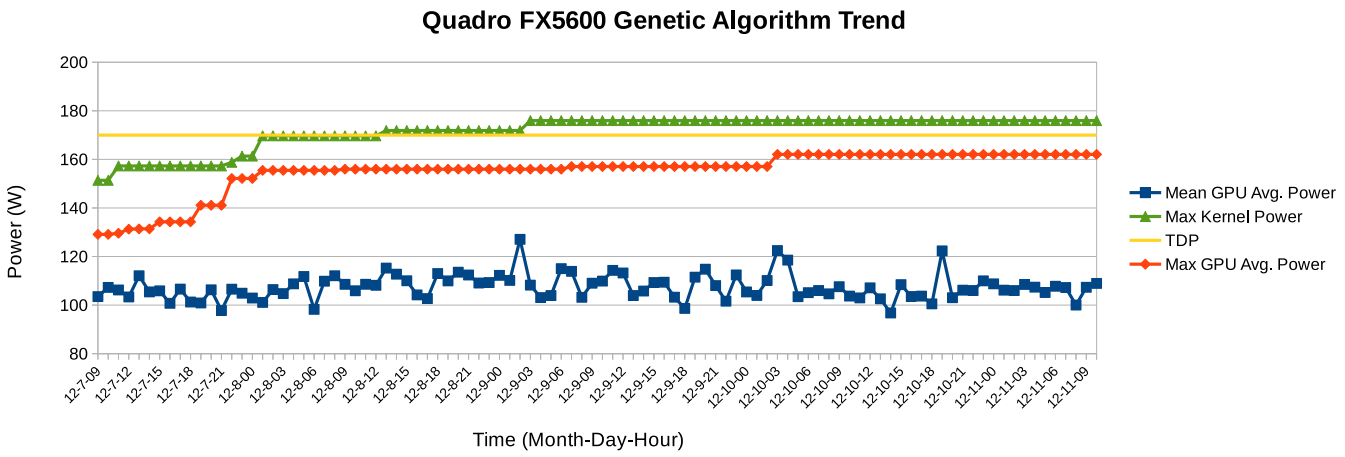

Fig. 6: Quadro FX5600 - GA Timeline

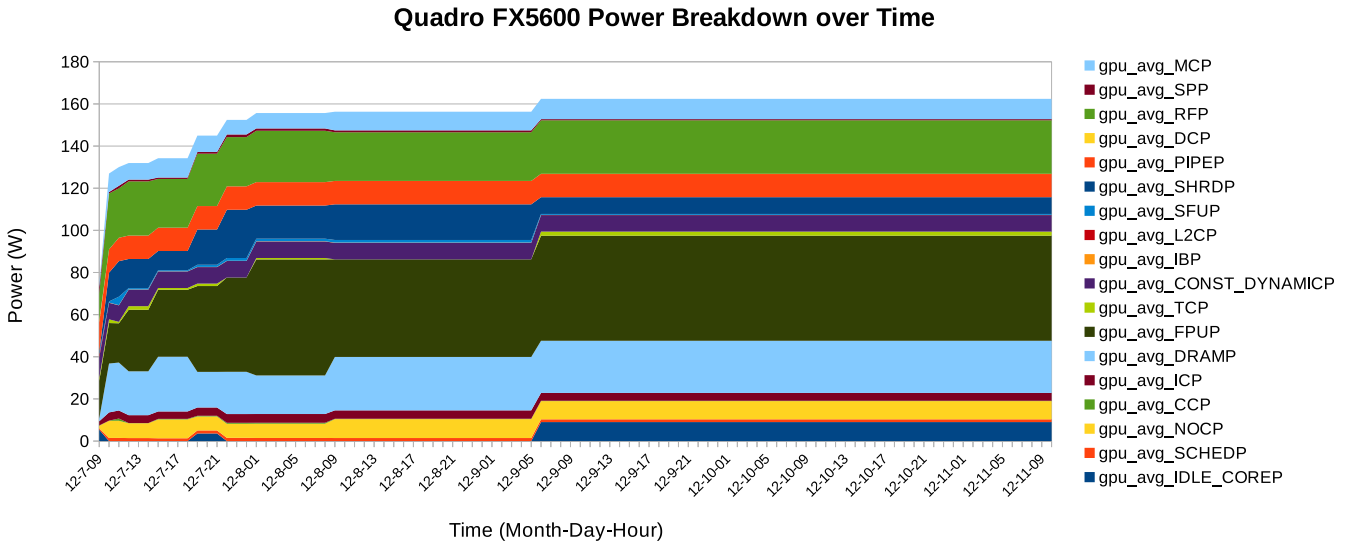

Fig. 7: Quadro FX5600 - GA Timeline Components

be caused by the ordering of different sets of instructions and operands that create unexpected contention.

We continued the seed variation analysis by varying the seeds for the Top 5 performing power viruses for each architecture with the results for the Quadro FX5600 shown in Figure 8. Once again we saw significant variations between individuals, but we also saw that the originally second best performing power virus produced the best result once we started varying it's seed. The amount of variation seems fairly steady between sets of individuals with only varying seeds, but we would need a bigger sample size to confirm.

This shows that seemingly minor differences between instruction and memory operand usage/placement can have a significant effect on the outcome; however, there also appears to be limits on how much the result varies. Configurations that can produce the best power viruses do not produce the worst results when given the wrong seed, just uninteresting results (a result of 120 W is still somewhat high just not very interesting for what we're looking for). In order to improve results, we would have to either add additional parameters or constraint that force the code generator to produce less variation between individuals with different seeds or run certain individuals multiple times to get a range of values to choose from. In either case this would significantly expand the search space and would need to be done in an intelligent manner. As it stands now, it does not seem like this limitation prevents us from finding high performing power viruses since there are likely numerous variations of these parameters that will give useful solutions, and it is unlikely that none of them would be created due to an unlucky RNG seed.

E. Trends for Analytical Modeling

The most helpful tool in discovering trends or for narrowing the search space has been Plackett and Burman analysis. As described earlier, we were able to prune large sections of the search space and identify coarse grained rules and heuristics

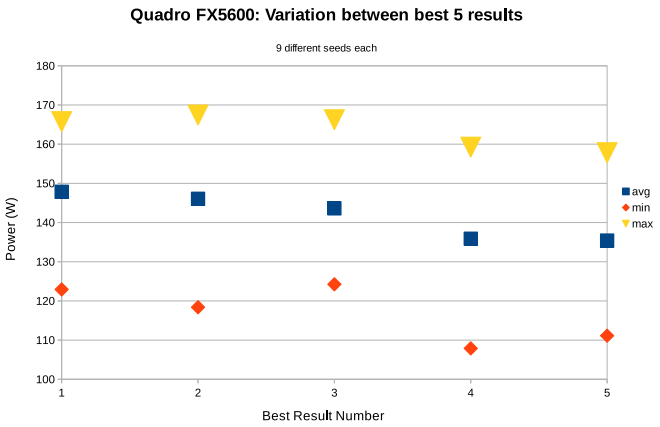

Fig. 8: Quadro FX5600 - Top 5 - Variation

for each architecture. When we ran the GTX 480 GA, we performed the PB design after we had already started the algorithm. Initially we used the analysis to modify our code generator to fix a few bugs that it helped identify and limited some of the GA's search space for areas that were not at all productive (limited it to FP only which cut down on some of the other types of instructions). The findings from that analysis were later mirrored in the best performing viruses. In previous work, Yi et al. [9] introduced using PB design when varying architectural parameters to determine what mattered most for improving performance. We found that the concept of creating a power virus mirrored that description well and helped focus the search space and confirm results. They also found that higher order effects from varying architectural parameters were generally low which lends more credibility to using PB for identifying the proper bounds to place on the search space. Any analytical model for generating the code generation parameters based off of architectural parameters would certainly benefit from testing/tuning those results with PB design. Additionally PB design can be very effective for ensuring that bounds you place on parameters for a GA are sufficiently wide to effectively explore the search space.

The primary trend we found was that having the right number of threads makes the biggest difference. As described in the GPGPU-Sim paper too few or too many threads can lead to poor performance and similarly too few or too many threads can lead to lower power usage. The number of threads does not directly correspond to any single architectural feature but it appears to be primarily affected by the warp width which is essentially the width of a processor. The Quadro FX5600 has a warp width of 8 threads and uses far fewer total threads; whereas, the GTX 480 has a warp width of 32 and uses more threads to create its top performing power viruses. Additionally the processors we tested used mostly floating point add, divide, and simple ALU instructions, with small distances between successive divide operations. It is unsurprising that floating point instructions use more power since they are more complex and generally complete in similar amounts of time. Additionally these architectures had a floating point unit for each thread in a warp. As long as a future architecture has similar ratios of latencies between FP and Int and between threads and FP units we should use an instruction mix that heavily uses floating point division and lesser amounts of addition, subtraction, multiplication, and other simple ALU instructions. These architectures also had fewer SFUs than threads per warp thus a warp could not be fully utilized if there was an SFU instruction. As such they showed much lower power usage whenever they used SFU instructions; however, the ratio of SFU power compared to what the approximate corresponding regular ALU instructions power seemed to favor SFU if equal numbers of each were concurrently ran (see the Monte Carlo benchmark as an example and keep in mind that the power usage of the SFU unit was with a 1 to 8 ratio to threads per warp) . As long as there are fewer SFUs then threads, a power virus should avoid using those instructions though that may change if the ratio between SFUs and threads per warp is 1 or 2. However, these ratios are likely to remain high since SFU instructions are not as frequently used. In both architectures shared memory was the most highly used memory operand. Shared memory on most devices has similar latencies to texture and constant memory and only slightly worse than registers. Unlike textures and constant memory, shared memory can be written to which adds to its complexity and likely power usage. It is likely that each GPU will get the most power usage from some mixture of register usage, global memory usage, and shared memory usage with shared memory being most heavily used and global memory the least (texture and constant memory used sparingly if at all) though significant changes in the relative latencies may change that.

VI. Future Work

The obvious next step would be to verify our results on a real GPU. The choice of using GPGPU-Sim was determined by what we had available to work with as opposed to what would give the best results. Verifying the results of the simulator would not only confirm that the algorithm works, but that it would work for architectures we only have a model of like an architecture underdevelopment. Additionally, with a large and more diverse set of GPUs we would be able to derive better relationships between architectural features and what the corresponding code generation parameters are. We may need to make radical changes to the GA if we discover significant new architectural features have been added (like allowing the ALU and SFU to run concurrently). At this point, regression models or PCA could be utilized to expose trends in the data and architectural parameters yielding the greatest variation. Furthermore, due to the (partially) random nature of the code generator, generated code needs to be analyzed in more detail and correlated with Code generation and architectural parameters.

Currently, the GA is tuned to coarsely explored the search space. While such an approach is useful for avoiding local maxima, it can be an issue when trying to tune a result. However, an adaptive GA would be able to change mutation/crossover rates, selection methods, and other aspects in order to cater to coarse and fine-grain exploration. Firstly, the mutation and crossover rates have artificially been set

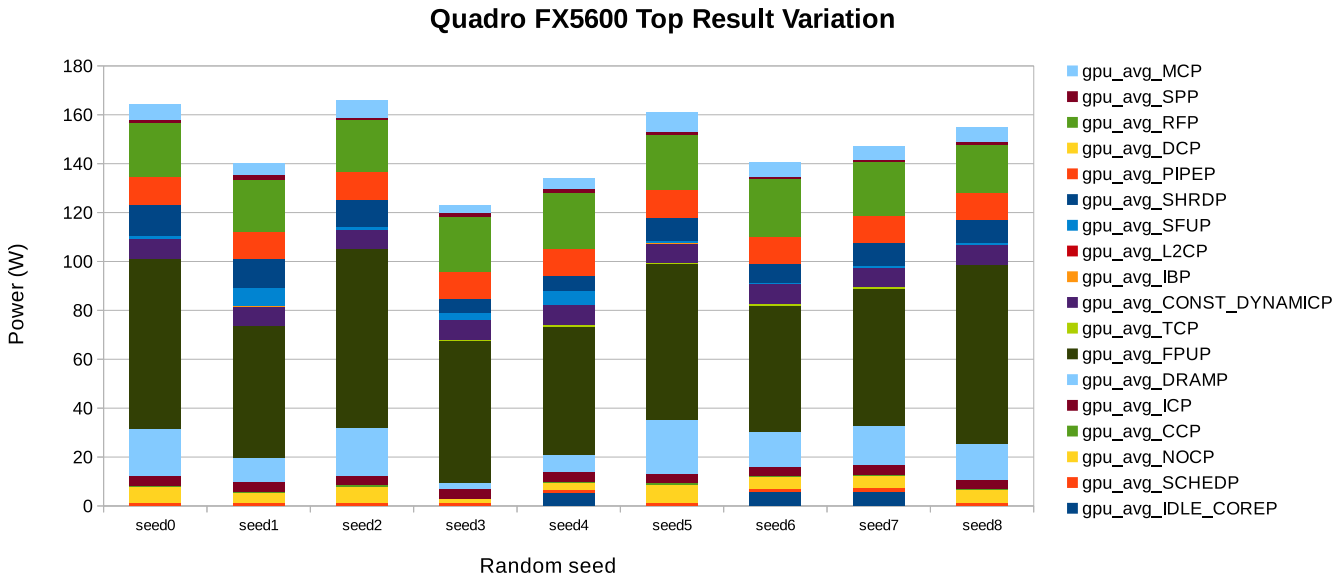

Fig. 9: Quadro FX5600 - Seed Variation

to a higher than normal value (40% and 30%, respectively). Initially, this can be acceptable or during spans of duplicate individuals (escape local maxima). However, as results start converging (defined at improvement/time), mutation and crossover rates should decrease. Additionally, second-order parameters such as the RNG seed value should be introduced. Effectively, this will allow for GA results to adequately explore the search space and finely develop top performing individuals. With this addition, the current population purge functionality could be removed, unless it was seen to still improve the diversity of individuals.

VII. Conclusion

GPUs are being increasingly used for non-graphical programs with high parallelization to enhance throughput over using just the CPU for the same work. One of the concerns shared by both CPUs and GPUs is power consumption at peak performance and how to develop programs that will allow developers to test their system when at or near that point. We created a tool to create power viruses for various GPU architectures using GPGPU-Sim and GPU-Wattch to extract the most power consumption from such devices. In the end, our novel distributed Genetic Algorithm was able to create power viruses for the GTX 480 and Quadro FX5600 that came close to or exceeded the thermal limits for these cards and discovered correlations between certain hardware features and the corresponding parameters to our code generator that get close to producing the best power viruses. We had some limitations for our analytical model with the architectural search space we were allowed, but managed to produce some interesting conclusions. Some of our most important findings are that thread count is most directly correlated to the total power consumption, simple floating point instruction with heavier weight towards division are desirable for higher power consumption. The most important memories to increase power consumption are shared, global, and register file. Furthermore, we found Plackett and Burman design to be most helpful for tuning the search space for our genetic algorithm and for discovering trends in our results.

References

[1] Norinder, Ludvig. ”Breeding power-viruses for ARM devices.” M.S. Thesis, 2013.

[2] Ganesan, K.; John, L.K., ”MAximum Multicore POwer (MAMPO) An automatic multithreaded synthetic power virus generation framework for multicore systems,“ *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pp.1,12, 12-18 Nov. 2011.

[3] K. Ganesan, J. Jo, L. W. Bircher, D. Kaseridis, Z. Yu, and L. K. John, ”System-level Max Power (SYMPO) - A Systematic Approach for Escalating System-Level Power Consumption using Synthetic Benchmarks,“ *in Nineteenth International Conference on International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Vienna, Austria, 2010.

[4] Bakhoda, A; Yuan, G.L.; Fung, W.W.L.; Wong, H.; Aamodt, T.M., ”Analyzing CUDA workloads using a detailed GPU simulator,” *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, vol., no., pp.163,174, 26-28 April 2009.

[5] Jingwen Leng; Tayler Hetherington; Ahmed ElTantawy; Syed Gilani; Nam Sung Kim; Tor M. Aamodt; Vijay Janapa Reddi. ”GPUWattch: enabling energy optimizations in GPGPUs,“ *SIGARCH Comput. Archit. News*, pp.487-498 June 2013.

[6] Reinman, G. and Jouppi, N. 1999. An integrated cache timing and power model. CACTI 2.0 Tech. Rep., COMPAQ Western Research Lab.

[7] Jieun Lim; Nagesh B. Lakshminarayana; Hyesoon Kim; William Song; Sudhakar Yalamanchili; Wonyong Sung, ”Power Modeling for GPU Architectures Using McPAT“, *ACM Trans. Des. Autom. Electron. Syst.* vol.19, no.26, June 2014.

[8] Franois-Michel De Rainville, Flix-Antoine Fortin, Marc-Andr Gardner, Marc Parizeau and Christian Gagn, ”DEAP: A Python Framework for Evolutionary Algorithms”, *EvoSoft Workshop, Companion proc. of the Genetic and Evolutionary Computation Conference (GECCO 2012)*, July 07-11 2012.

[9] Yi, J.J.; Lilja, D.J.; Hawkins, D.M., ”A statistically rigorous approach for improving simulation methodology,” *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, vol., no., pp.281,291, 8-12 Feb. 2003.

[10] Shuai Che; Boyer, M.; Jiayuan Meng; Tarjan, D.; Sheaffer, J.W.; Sang-Ha Lee; Skadron, K., "Rodinia: A benchmark suite for heterogeneous computing," *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp.44,54, 4-6 Oct. 2009.

APPENDIX A
GPU ARCHITECTURAL PARAMTERS

TABLE II: Code Generation Parameters

Parameter	**GTX480**	**Quadro FX5600**
CUDA Cores	448	128
SIMD Width	32	8
Graphics Clock	607 MHz	600 MHz
Processor Clock	1215 MHz	600 MHz
Memory Clock	1674 MHz	800 MHz
FP Units	32	8
INT Units	32	1
SFU Units	8	1
L2	Yes	No