GPU and CPU Parallelization of Honest-but-Curious Secure Two-Party Computation

Nathaniel Husted
Dept. of Comp. Sc.
Indiana University
nhusted@cs.indiana.edu

Steven Myers
Dept. of Comp. Sc.
Indiana University
samyers@cs.indiana.edu

abhi shelat
Dept. of Comp. Sc.
University of Virginia
abhi@cs.virginia.edu

Paul Grubbs
Dept. of Comp. Sc.
Indiana University
paulgrub@umail.iu.edu

ABSTRACT

Recent work demonstrates the feasibility and practical use of secure two-party computation [5, 9, 15, 23]. In this work, we present the first Graphical Processing Unit (GPU)-optimized implementation of an optimized Yao's garbled-circuit protocol for two-party secure computation in the honest-but-curious and 1-bit-leaked malicious models. We implement nearly all of the modern protocol advancements, such as Free-XOR, Pipelining, and OT extension. Our implementation is the first allowing entire circuits to be generated concurrently, and makes use of a modification of the XOR technique so that circuit generation is optimized for implementation on SIMD architectures of GPUs. In our best cases we generate about 75 million gates per second and we exceed the state of the art performance metrics on modern CPU systems by a factor of about 200, and GPU systems by about a factor of 2.3. While many recent works on garbled circuits exploit the embarrassingly parallel nature of many tasks that are part of a secure computation protocol, we show that there are still various forms and levels of parallelization that may yet improve the performance of these protocols. In particular, we highlight that implementations on the SIMD architecture of modern GPUs require significantly different approaches than the general purpose MIMD architecture of multi-core CPUs, which again differ from the needs of parallelizing on compute clusters. Additionally, modifications to the security models for many common protocols have large effects on reasonable parallel architectures for implementation.

*This work is supported by Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contract FA8750-11-2-0211. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US government. This material is based upon work supported by the National Science Foundation under Grant No. 1111149.

ACSAC '13 Dec. 9-13, 2013, New Orleans, Louisiana USA

http://dx.doi.org/10.1145/2523649.2523681.

1. INTRODUCTION

A company may wish to offer a generic screening service which would let patients know if they are susceptible to disease based on the presence of different proprietary markers in their DNA. In such a scenario the company does not want to divulge its proprietary markers, and the consumer does not want to divulge their genetic information in fear that it will be exploited by the company. The above problem represents a specific case of *secure two-party computation*, in which there are two parties who wish to compute a function $f : (\{0,1\}^m)^2 \to \{0,1\}^m$, on respective inputs $x_0, x_1 \in \{0,1\}^m$, with the guarantee that no party learns anything beyond what can be efficiently inferred from the output.

Cryptographers have studied this problem, and have suggested solutions in various security models. However, while theoretically interesting, it was historically believed that these protocols are too inefficient for practical implementation. Work by Malkhi et al. [18] gave the first implementation of Yao's garbled-circuit protocol and, while it could perform very modest computations in a reasonable amount of time, the result seemed to validate the belief that these protocols would not be practical. This has resulted in cryptographers pursuing specific protocols to solve specific instances of secure two-party computation. For example, specific algorithms for looking at the edit distance between two strings (e.g., for the genetic problem discussed above), with the goal of making practical algorithms that could be deployed. However, recent advances and improvements to Yao-based protocols and implementations (cf. [13, 21, 8]) have shattered the belief that general purpose solutions are too inefficient to be deployed in practice. These works have lead to renewed interest in practical implementations of Yao's protocol. Research now focuses on determining which problems might be solved by efficiently engineered versions of Yao's garbled circuits. Any such engineering will make use of parallel processing, as Yao's circuit protocol (and its improvements), have a high level of inherent parallelism for both the honest-but-curious and malicious security models. Importantly, there are key differences in the available parallelism available in the two security models.

In this paper, we provide a high-performance parallel implementation of Yao's circuits in the *honest-but-curious (HbC) and 1-bit-leaked malicious model (1BM).* The implementation is optimized for parallel processing architectures with both multi-core CPUs and GPUs. We have implemented both circuit generation and evaluation on GPUs. Additionally, on multi-core CPUs we

have implemented evaluation. GPUs have shown to provide more GFLOPS per dollar and more GFLOPS per watt than leading x86 CPUs, and the gap in performance is expected to widen. Therefore, any compute intensive task that can naturally be parallelized, such as Yao's garbled circuit technique, needs to be investigated in this model. Further, it has previously been noted that GPUs can potentially be used as cheap, "off-the-shelf" cryptographic co-processors, and work has been done showing their use for implementing both symmetric and asymmetric cryptographic primitives, as well as for cryptanalysis.

Frederiksen and Nielsen[5] have recently produced a somewhat optimized version of Yao's protocol in the *malicious security model* for GPUs. However, differences in Yao's protocol for the malicious model, as compared to the ones we consider, necessitate different architectural approaches for implementation. This is particularly poignant due to the the Single Instruction Multiple Data (SIMD) architecture of the GPU: *small changes in protocols can lead to large changes in the appropriate processing units for a GPU*. Thus, our work is important as it provides a new GPU work scheduling architecture optimized for the HbC and 1BM security models, which have many practical deployment scenarios.

1.1 Our Contributions

We present the first modern implementation of Yao's for the Honest-but-Curious (HbC) and One-bit Leak Malicious (1BM) security models. There are many settings in practice, where these security models are more than sufficient for use in secure computation. Due to the fact that protocols that satisfy these weaker security models are substantially less resource demanding, it permits for either a larger array of circuits to be evaluated, or for systems to potentially be much faster.

In this paper, *we first present a new method to generate garbled circuits with Free-XORs so that generation can be entirely parallelized on the GPU.* Prior works have parallelized the generation of "layers" of the circuit, but suffered from inherent data dependencies that prevented parallelizing the generation of the entire circuit due to the Free-XOR optimization technique. By default, garbled circuits are not designed to be optimal for GPUs and SIMD computation when using the Free-XOR technique because this technique creates dependencies in XOR gate chains. A principle contribution of this work is to fix this issue at the protocol level; we are the first implementation to do so and our experimental results validate the benefits of the approach. Our resulting GPU-based implementation provides significant improvements in circuit generation speed, compared to all previous constructions, both those using GPUs and those that do not. We provide a more detailed explanation of our system in Sec. 7 and additionally discuss implementation issues for the GPU and some simple optimizations to implementations of SHA1.

Next we see that for our security models the evaluation of circuits on a relatively simple CPU implementation outperforms our GPU implementation. Therefore, any system in which the GPU is going to be used to maximal capacity for evaluation is going to need to improve the ability to fully parallelize evaluation. Specifically, evaluation of a garbled circuit has an inherent data dependency between layers; one cannot evaluate layer i of the circuit without having first evaluated all the gates which it depends on layer $i-1$. A strong deployment will need to have a finer understanding of dependency that allows for gates in higher levels to be evaluated (if possible), before all lower level gates are evaluated. Further, our experiments suggest a reasonable architecture for Yao circuits in the HbC and 1bM models may also involve a hybrid approach: use the GPU for generation and verification and then splitting evaluation between the CPU and GPU.

Overall our performance results are better or comparable to the other state of the art implementations, with results varying on the facets of circuit generation or evaluation considered. We show we can generate gates using the GPU significantly faster than the only other GPU implementation of Yao's [5], and on a per system basis we can generate and evaluate gates on CPUs faster than Kreuter et al. [15]. Specifically, on similar top-end hardware from 2013, our system can generate roughly 74 million gates/sec, whereas [5] achieves 21 million gates/sec and Kreuter et al. achieves 0.35 million gates/sec.

1.2 Roadmap

In Section 2 we present related work. Section 3 gives a brief overview of the different security models discussed in the paper, and of Yao's protocol and its variants. In Sections 4 and 5 we briefly introduce and discuss some of the architectural issues in GPU development and multi-core CPU development respectively. In Section 6, we discuss how the different security models induce different architectural approaches to accommodate the different types of parallelism in the underlying protocols and the resources they use. In Section 7 we detail how our system works, and our modification to the Free-XOR technique that allows for faster circuit generation. In Section 8, we provide experimental results validating our claims. Section 9 gives conclusions and discusses our current directions with this work.

2. RELATED WORK

Malkhi et al.[18] describe the first secure multi-party scheme implementing garbled circuits in the Fairplay system. Their system uses a custom circuit definition language called SFDL compiled into a machine-readable representation language called SHDL.

The first paper to consider the feasibility in practice of these schemes was Pinkas et al. [22], who implemented the first Free-XOR scheme [13] and OT Extension, and introduced the notion of Garbled Row Reduction to save communication costs. They also considered all modern security models. Huang et al. [8] improved Pinkas' et al.'s performance by utilizing a number of enhanced construction techniques for garbled circuits including Free-XOR, oblivious-transfer extension [11], the Naor-Pinkas OT protocol [20], and introducing the notion of pipelined gate generation and evaluation. The system still used serial gate generation and evaluation, but the authors showed the potential performance benefits of well-crafted circuits. Finally, the scheme was implemented in Java which is highly portable, but suffers from the need to be run through the virtual machine.

Pu et al. [23] gave the first implementation of Yao's circuits using a GPU. However, their implementation only used the GPU as a cryptographic co-processor to calculate symmetric encryptions (i.e. 3DES) and elliptic curve operations. *They do not attempt to use the GPU to actually build or evaluate any of the garbled circuits.* The system does not implement any of the modern algorithmic advances in garbled circuits such as Free-XORs, Pipelining, OT-extension, etc. The implementation is in the HbC security model.

Kreuter et al. [15] presented the first garbled circuit protocol that is secure against malicious adversaries and can scale to handle circuits with several billion gates. They implement all of the modern efficiency improvements to Yao's protocol, such as Free-XOR, Pipelining, OT-extension, etc. They also introduced a circuit compiler that translated C-like code into circuits and both the compiler and Yao's system could scale to handle several billion gates.

Huang et al.[9]present the first efficient protocol and implementation of Yao's in the 1BM model suggested by Franklin and Mohassel [19], but do not consider a parallelized implementation. They implement the Free-XOR technique, garbled row reduction and pipelining.

Frederiksen and Nielsen [5] have recently implemented Yao's on the GPU in the malicious model. Unlike the work in [23], they use the GPU not only to compute cryptographic primitives efficiently, but to generate and evaluate the circuit. They also implement modern efficiency improvements such as Free-XOR, garbled gate row reduction, OT extension. They do not implement pipelining.

Our approach is similar to the latter two works in that we exploit the parallel nature of certain subproblems of garbled circuits. However, we target the HbC and 1BM security models and thus the protocols we are implementing have less inherent parallelism. This is because the malicious model adds another layer of an inherently parallelizable protocol. Thus Kreuter et al. accelerate the cut-and-choose technique in the malicious model by giving each thread (i.e., processor) a circuit. They use a large compute clusters with hundreds of nodes to run their system. In particular, their MPI implementation is optimized for a *specific type of cluster*. Their code assumes the cluster's scheduler allocates work at the granularity of processors. Modern, high-end super computing clusters schedule at the granularity of nodes thus will not work optimally with their code. Similarly, Frederiksen and Nielsen also generate each copy of a circuit's gate in the cut-and-choose protocol on a separate GPU core.

In contrast, our system parallelizes the generation of circuits themselves, thus each core generates distinct gates of the circuit. Indeed, doing so requires changes to the protocol itself, but nonetheless, our systems are highly complementary, and a garbled circuit implementation that is a hybrid of our techniques could provide high performance. Further, because of the differences in the protocol, communication overhead in the HBC or 1BM security model is approximately 2 orders of magnitude less than these two prior works. This means that circuit generation and evaluation times are of prime importance, as compared to communication overhead as Frederiksen and Nielsen observe for the malicious model.

While there are competing approaches for constructing two-party secure computation protocols, it appears that the Yao garbled circuits approach is currently one of the fore-runner's in performance, although the recent SPDZ system of Damgard et al.[3] performs efficiently on some forms of circuit (such as, importantly, AES). Still, Major questions remain on how to optimize Yao's garbled circuits for speed depending on different compute models, and different security models. Solutions have currently focused on four approaches: i) Implementation Optimizations (Parallelism, pipelining), ii) Security Model Compromises (Hybrid Model), iii) Construction Optimizations (Free-XOR technique, OT Extension), and iv) Compiler Optimizations (Maximize XOR gates, minimize gate counts). This work focuses on implementation optimizations using the current best security model optimizations, construction optimizations, and compiler optimizations from the state of the art work. Specifically, we focus on parallelizing the generation and evaluation of garbled circuits so as to perform well on *single machines with GPUs*.

3. BACKGROUND

We (briefly) review Yao's garbled-circuit approach to secure computation [25], and the respective security models of interest: honest-but-curious (HbC), malicious, and one-bit leaked malicious (1BM).

Garbled circuits provide a way for two parties, holding inputs x and y respectively, to compute an arbitrary function f of their inputs without revealing anything to either party other than the result $f(x, y)$. At a high level, the idea behind the base protocol is that one party—the *garbled-circuit generator*—prepares an "encrypted" version of a boolean circuit for f (the *garbled circuit*) and sends it, along with an "encryption" of its input (say, x), to the second party. This other party—the *garbled-circuit evaluator*—obtains some additional information from the garbled-circuit generator (this information depends on the evaluator's input y), and then obliviously computes the output value $f(x, y)$ without learning the values on any intermediate wires of the circuit.

Security Models.

In the *honest-but-curious* model, both parties execute a protocol correctly, but the parties are willing to try and extract any extra information they can from protocol execution in other external processes. Thus, informally, a secure protocol must ensure that no extra information other than the output can be extracted or deduced in polynomial time from the protocol transcripts. A secure protocol in the *malicious* model is one that ensures the same even when an adversary deviates arbitrarily from the protocols specification. The prior two models apply rather generically to many cryptographic protocols. The final model is the *one-bit-leaked malicious* model. A secure computation protocol is secure in this model, if it is secure against malicious adversaries with the relaxation that an arbitrary predicate of the private inputs can be leaked to the adversary during any execution. Formal definitions of each of these models can be found in [6, 9].

All three of the models have legitimate practical scenarios. For example, hospitals might, in order to preserve privacy as dictated by law, determine if they have patients in common using the honest-but-curious model. Whereas, its use to securely compute in the presence of an adversary, such as a nation's intelligence bureau, would require malicious security. Companies might use secure computation in the one-bit-leaked malicious model, when the computation is not repeated frequently, and when no bit of the data is particularly valuable.

Yao's Protocol for the Honest-but-Curious Setting.

Given a boolean circuit for f (pre-agreed upon by the parties), the circuit generator chooses two random cryptographic keys W_i^0, W_i^1 for each wire i of the circuit. (The semantics are that W_i^0 encodes a 0-bit on the ith wire, while W_i^1 encodes a 1-bit.) In addition, for each wire i he chooses a random permutation bit π_i, and assigns key W_i^b the label $\lambda_i^b = b \oplus \pi_i$. Next, for each binary gate g of the circuit, having input wires i, j and output wire k, the circuit generator computes a *garbled gate* consisting of the following four ciphertexts (in order):

$$\begin{aligned}
&\mathsf{Enc}^g_{W_i^{\pi_i}, W_j^{\pi_j}}\left(W_k^{g(\pi_i,\pi_j)} \| \lambda_k^{g(\pi_i,\pi_j)}\right)\\
&\mathsf{Enc}^g_{W_i^{\pi_i}, W_j^{1\oplus\pi_j}}\left(W_k^{g(\pi_i,1\oplus\pi_j)} \| \lambda_k^{g(\pi_i,1\oplus\pi_j)}\right)\\
&\mathsf{Enc}^g_{W_i^{1\oplus\pi_i}, W_j^{\pi_j}}\left(W_k^{g(1\oplus\pi_i,\pi_j)} \| \lambda_k^{g(1\oplus\pi_i,\pi_j)}\right)\\
&\mathsf{Enc}^g_{W_i^{1\oplus\pi_i}, W_j^{1\oplus\pi_j}}\left(W_k^{g(1\oplus\pi_i,1\oplus\pi_j)} \| \lambda_k^{g(1\oplus\pi_i,1\oplus\pi_j)}\right),
\end{aligned}$$

where $\mathsf{Enc}^g_{W,W'}(m)$ denotes symmetric-key encryption of plaintext m using two keys W, W'. In our implementation, we define encryption as

$$\mathsf{Enc}^g_{W,W'}(m) = H(g\|W\|W') \oplus m,$$

where H represents a cryptographic hash function (SHA1) whose output is truncated to the length of the given plaintext. The set of all the garbled gates constitutes the garbled circuit that is sent to the evaluator. In addition, the circuit generator sends the permutation bit π_i for any output wire i of the circuit.

To evaluate the garbled circuit, the circuit evaluator must obtain keys for each input wire of the circuit; specifically, if the input bit on some input wire i is b_i, then the evaluator should be given the key $W_i^{b_i}$ along with the label $\lambda_i^{b_i}$. (Furthermore, for each input wire it should get *only* that key.) For input wires that correspond to the input of the circuit generator, x, this can easily be arranged by simply having the generator send the appropriate key/label for each such wire. The parties can use *oblivious transfer* [6] to allow the evaluator to obliviously learn the appropriate keys/labels for input wires that correspond to its own input, y.

Given keys/labels W_i, λ_i and W_j, λ_j associated with the input wires i, j of some gate g, the evaluator can compute a key/label for the output wire of that gate by decrypting the ciphertext in position (λ_i, λ_j) of the garbled table for g. Proceeding in this way through the entire (garbled) circuit in topological order, the evaluator can compute keys/labels for each output wire of the circuit. Using the permutation bits sent to him by the circuit generator, this means that the evaluator can determine the actual bit output of the circuit. If specified as part of the protocol, the evaluator can send that result back to the other party.

The crucial point for our purposes is that generation of all the garbled gates can be done in parallel once the wire keys and permutation bits have been chosen. Further details, along with a proof of security, can be found in [17].

Malicious Security Model: Cut-and-Choose.

There are several known ways to modify Yao's protocol to the malicious model, but the only one that has been implemented and deemed practical is the cut-and-choose approach. Here the circuit generator creates not one 'encrypted' circuit, but $\approx k$ 'encrypted' copies of the circuit (where k is a security parameter). The generator sends the k encrypted circuits (without the encryption of its input to the generator). The evaluator now chooses $\approx 60\%$ of the circuits to be revealed, at which point the generators gives all the information necessary to generate the circuit to the evaluator who then verifies that all the circuits are legitimate implementations of the correct circuit for f. If not, the evaluator quits. If so, the evaluator asks for the "encrypted" inputs to the remaining $\approx 40\%$ of the circuits, and computes the output. The evaluator takes as its output the majority output of the many evaluated circuits. The argument for the security of this protocols is beyond the scope of this paper but can be found in many places (cf. [24]).

One Bit Leaked Malicious Model.

In this model, the protocol is modified so that each party plays *both* the role of generator and evaluator. Each party generates a circuit and sends it to the other, which in turn evaluates it. After this is done, a specialized protocol *that is secure in the traditional malicious model* does a secure function evaluation to ensure that the outputs of both evaluated circuits are the same. A specialized and efficient protocol (both in computation and communication) for verifying the equality of outputs in the malicious security model is given by Huang et al. [9].

4. GPU COMPUTING AND CUDA

Modern GPUs are massively parallel computational devices, but differ from modern multi-core CPUs in significant aspects. In this section we provide a brief overview of their architecture. Communication to and from the GPU occurs over the system PCI bus, which is substantially slower than the regular communication path between RAM and the CPU on a modern machine.

Anatomy of a CUDA GPU.

The smallest execution unit on a CUDA[1] GPU is called a streaming processor (SP), or CUDA core, which is capable of executing an independent thread. These cores are not equivalent to CPU cores but more equivalent to lanes on a vector processor. An SP has access to local memory and registers. Multiple SPs are combined to construct one Streaming Multiprocessor (SM). The number of SPs located in an SM, and complexity of the SM depend upon the GPU hardware generation. Every NVIDIA GPU has multiple SMs.

SMs are in charge of scheduling work to their SPs. SM's receive work in the form of thread blocks. Thread blocks can contain a number of threads defined by the programmer at run time. The SM then splits these thread blocks into groups of 32 threads called warps. Each warp is run on a set of 32 SPs. Each thread in a warp executes one common instruction at a time thus warps are akin to 32-wide vector processors. To ensure every thread is executing the exact same instruction, each thread in a warp must execute the exact same branch in program flow. If different threads need to run different branches, the GPU serializes them by having the appropriate threads execute the branch, while non-branching threads' processors sit idle. When such a divergence in execution occurs between threads in a warp it is termed *warp divergence*. *To get full efficiency from the GPU it is essential that programs be written so that they can be broken down into warps where all threads execute the same instruction*, and there is little to no conditional branching that does not affect all the threads in the same manner. This is the *Single Instruction Multiple Data* (SIMD) paradigm for parallel programming, where the same instruction is applied to multiple pieces of data at a time.

Also note that a given SM might be concurrently executing multiple warps via "hyper-threading". If one warp is waiting on memory access or some other condition, the SM may start to execute another warp. There is little to no cost in time for this context switching, however it does mean that local resources such as registers and local memory need to be shared between these warps.

Relative Speeds of Memory and CPU-GPU bandwidth.

GPUs must deal with latency issues caused by transferring data from the host machine to the GPU and vice-versa. Transfer rates between the host and GPU are $\approx$ 8GB/s over the PCI-E bus when the GPU is on a PCIe card. The transfer rates are orders of magnitude slower than the GPU's theoretical compute throughput. Memory transfer on-board the GPU is several orders of magnitude faster than the bus (e.g., global memory on a Tesla card transfers at 177.6 GB/s). Therefore, high performance requires minimizing memory transfers and the communication between the GPU and CPU, and maximizing the local computation performed on the GPU. Differing types of memory on the GPU, including global, shared, local (L1, L2 cache), and registers, also operate at varying speeds and thus create a memory hierarchy on the GPU mirroring that on a typical machine. Kernels must optimize the use of local registers and shared memory while dealing with the extremely limited resources of each. Register dependencies, such as when a read directly follows a write to the same register, can also increase latency. Thus

[1]By CUDA (Compute Unified Device Architecture) we mean an NVIDIA GPU that supports NVIDIA's CUDA programming environment. The most commonly developed for GPU.

we want to maximize register usage to increase speed, however any particular thread also wants to minimize usage so that a SM can "hyper-thread" multiple warps, if any of them are latent due to memory fetches or other reasons. These conflicting goals make register usage a complicated trade-off in GPU programming.

5. MULTI-CORE CPU VS CLUSTERS

For our CPU based circuit evaluation system, we use local parallelism to take advantage of the multi-core environment found on modern day CPUs. This is in contrast to the different, if not complimentary, approach taken by Kreuter et al. [15], who take advantage of the parallelism available on multi-node compute clusters. We discuss the different technological approaches next.

MPI and OpenMP.

OpenMP and MPI (Message Passing Interface) are both competing and complementary standards for parallelization in High Performance Computing (HPC). OpenMPI is currently a dominant MPI implementation but is not related to OpenMP beyond being a parallelization technology.

MPI is a message passing technology that enables "scale-out" parallelism on multi-device compute clusters, such as super computing clusters. The developer defines an MPI process that is launched many times on many different compute nodes. These MPI processes are able to pass messages between one another when they need to share computation. It works best for large jobs on large systems, as each MPI process incurs large overhead. MPI requires that any machines running MPI code have an MPI implementation's executables installed, for example, Open MPI's libraries and executables.[2] OpenMPI is the technology used by Kreuter et al.[15].

In contrast, OpenMP is an HPC technology designed for "scale-up" parallelism on a single machine. It is a standard that is built in to compilers such as GCC and includes a small driver library. Developers use OpenMP to easily create many lightweight threads with minimal syntax compared to traditional POSIX thread implementations. Unlike POSIX threads, OpenMP is optimized for a data parallel programming paradigm and not a task parallel programming paradigm. Data parallelization is akin to the SIMD (Single Instruction Multiple Data) paradigm and task parallelization is akin to the MIMD (Multiple Instruction Multiple Data) paradigm. OpenMP is the technology we use for our parallel multi-core CPU evaluation scheme.

6. SECURITY MODEL INDUCED ARCHITECTURE TRADE-OFFS

While GPUs gain with massive parallelism, they lose in terms of algorithmic flexibility. Programmers must specify the logical allocation of their threads in terms of thread blocks, and these thread blocks affect physical GPU allocation. If this logical allocation is poor (e.g., setting thread blocks to have one thread) then poor performance follows, as many cores in a SM sit idle while a single core computes the thread. We compare the architectural approach of Frederiksen and Nielsen [5] and Kreuter at al. [15] given their malicious model implementations, and our approach in the HbC and 1BM security models. Recall that in the cut-and-choose malicious implementation the generator must generate $\approx k$ circuits while the evaluator will evaluate some 40% of those circuits, and verify the remaining 60% to ensure they were properly constructed. In the HbC case the generator must generate only one circuit and the evaluator must evaluate only one circuit. In the 1BM case, each party must generate and evaluate one circuit.

[2] `http://www.open-mpi.org/software/ompi/v1.6/`

Similarity between HbC and 1BM.

Implementation details between HbC and 1BM protocols are generally identical as the resources they need are very similar. The 1BM protocol differs in only requiring one more circuit generation and one more circuit evaluation than that of the HbC protocol. Therefore, we address both protocols in our discussion as if they were the same model.

Communication Differences.

One immediate observation is that in the malicious model, reasonable values of k might vary between 60 and 120, and thus the number of circuits that need to be transferred between the two agents in the protocol 40% of this, [3] compared to the one or two circuits that need to be transmitted our protocols. Frederiksen and Nielsen show that in their protocol with varying security parameters, that communication costs dominate, often by a factor of 3 to 4 times the generation or evaluation times. This is not by enough that one should expect them to dominate in the HbC or 1BM scenario we implement. Further, recent advances in the cut-and-choose methodology by Lindell [16] and optimizations that Frederiksen and Nielsen did not implement [24], further reduce the communication burden.[4] Finally, Frederiksen and Nielsen also increase the circuit-size to include a universal hash of the inputs, and there are alternate approaches that will not increase the circuit size which can be considered. Therefore, we can consider optimizations that disallow the garbled row-reduction methodology, and also slightly increase communication for the sake of efficiency.

Malicious Cut-and-Choose and the GPU.

Cut-and-choose protocols must generate k circuits at a time. Instead of having each thread represent a gate, in cut-and-choose each thread represents one of the k circuits and the thread block is used to represent an individual gate. Thus, each thread block contains k threads and each thread deals with the generation of a specific gate for each of the k circuits. One can then allocate the same number of thread blocks as there are gates in the circuit to the GPU. As each thread block will always have threads processing the same gate type, there is no fear of warp divergence. The only caveat is the thread block size, and thus the cut-and-choose security parameter, should be a multiple of 32 for optimal GPU allocation (again, the SMs allocate 32 threads at a time). Levels are evaluated in turn, but this is less problematic to performance, circuit widths are effectively multiplied by k, meaning the GPU spends little relative time idle waiting for a level to complete before the next is started. Evaluation occurs in a similar manner, but must use the level-by-level process that was used in semi-honest evaluation. For each level there will be a thread block for each gate in that circuit's level and each thread block will contain k threads. We note this is exactly the approach taken by Frederiksen and Nielsen [5].

Honest-but-Curious and One-Bit Malicious.

The previous description of a successful approach to placing cut-and-choose on the GPU should make it clear why the same approach is inefficient for the semi-honest case (and similarly the 1BM case). If only one circuit is being generated or evaluated, each thread block will contain only one thread, and only one thread will be allocated by the SMs on the GPU for each gate leaving 31/32

[3] It is known that *only* the circuits that are being evaluated need to be transferred as noted in [7], as it suffices to communicate a collision resistant hash of the verified circuits.

[4] In practice Frederiksen and Nielsen evaluate 50% of the circuits.

SPs in a warp dormant (meaning the vast majority of GPU cores are consistently going unused).

We describe our approach to implementing HbC and 1BM on the GPU. For simplicity we will assume the circuit we are generating and evaluating fits in GPU memory, although our approach is not limited in this fashion. As we are only concerned with a single circuit, we will pass the whole circuit description to the GPU for generation and evaluation. In the case of generation, each thread represents a single gate in the circuit and the number of threads allocated are the number of gates in the circuit. The size of a thread block does not matter for the HbC or 1BM case, although for efficiency it should be a multiple of 32 (the physical thread allocation count by the SM). We can handle XOR gates with one kernel and all other truth table gates with another kernel. The latter essentially always make a blank truth table, and the converts it to the appropriate type by changing each line of the table to describe the appropriate operator. However, to construct a truth table gate one needs to have knowledge of its input wires' labels. This too can seemingly be solved because we can pseudo-randomly generate a wire's labels based on the wire's identifier, but this conflicts with the Free-XOR technique (as described in the next section). We modify the Free-XOR technique at the cost of a small amount of extra communication, and allow all gates in the circuit to be generated in parallel, independent of the level on which they reside.

Evaluation needs to occur on a level-by-level basis to honor data dependencies between gates. Again,we would have two kernels for evaluating the two types of gates, truth table and XOR. Consider what happens when evaluating the end of a level: it is likely that many symmetric processors will sit idle waiting for the level to be completed, as there are no more gates on the current level, say $i-1$, to evaluate, but they cannot commence processing the level i gates until the last few gates on level $i-1$ are evaluated due to potential dependency issues. The narrower a level is, the larger the inefficiency if many of the GPU's cores need to lay latent why completing the level. Logic to check for dependencies is likely to cause divergence, and latency problems. Therefore, for circuits which are not wide, the relative amount of time that is spent by the cores being idle at the end of a level can be quite large. In the cut-and-choose approach, the fact that there are k copies of each circuit has the effect of essentially multiplying the width of each circuit by k, making the issue far less problematic. Of course, for particularly large and wide circuits, this should not cause much of an issue for the HbC or 1BM implementations.

7. ARCHITECTURE & METHODOLOGY

Several optimizations of Yao's garbled-circuit protocol have been proposed, but it is not clear how all of them can be efficiently implemented in a massively parallel system. Here we discuss the major techniques and our approach to implementing them. As a starting point, we implement the folklore "single row evaluation" technique already described in the description of the Yao protocol in Section 3. This optimization, created by [18], decreases evaluation time on encrypted gates by roughly a factor of 4.

On the other hand, one popular technique for reducing the size of garbled tables by $1/4$, called *Garbled-row reduction* [21], is not implemented as any such implementation would seem to slow execution on a SIMD parallelization.[5] The benefit of this approach is that it reduces communication by 25%, but as discussed our security models prompts us to be less concerned about communication costs and more about gate generation and evaluation timings.

7.1 Selection of the Random-Oracle/Permutation Function

The Yao-garbled circuit technique relies on symmetric encryption. In most modern implementations the symmetric encryption is provided via a random oracle instantiated by a cryptographic hash (SHA1). Recent work by Bellare et al. [1] has also considered the use of a Random Permutation that can be instated with fixed key in a block-cipher such as AES. In our implementation, we choose the SHA1 function for this purpose. Jang et al. [12] showed that AES had substantially slower throughput than SHA1 on GPU architectures. We have not had the chance to consider an optimized version of AES with a fixed key, as suggested in [1], and this should be investigated. We experimentally tested BLAKE256, a SHA3 finalist, which also had a slower throughput on the GPU than SHA1, when both are given inputs that require the same number of rounds of Merkle-Damgard.Our implementation of SHA1 is a hand optimized version of the John the Ripper implementation of SHA1.[6] In particular, we are able to reduce the number of rounds we typically need to calculate in SHA1 by judiciously choosing the values we hash. As others have done, we ensure that we never need to hash more than one block of data. However, *by ensuring that the prefix of the values we hash for a given circuit remain relatively constant, we can pre-compute the first few rounds of SHA1 for each query in generating or evaluating a circuit.* This allows us to knock off either 6 or 14 rounds (out of the 80 rounds) of each SHA1 call. We also note that to allow an SM to be able to "hyper-thread", we need to be miserly with our use of registers in this code. Profiling clearly shows that hand optimization of the SHA1 code is a worthwhile endeavor.

7.2 GPUs and Free-XOR

One main challenge in this work is to develop a version of the Free-XOR technique that is compatible with parallelization. We begin by describing the technique.

The *Free-XOR* technique [13] allows XOR gates in the circuit to be evaluated using only a bit-wise XOR operation instead of the standard garbled-gate evaluation. In this approach, the circuit generator chooses a global random offset R, and then ensures that the keys for every wire i in the circuit satisfy $W_i^0 \oplus W_i^1 = R$. This is usually done by choosing keys for each wire i in the circuit that is *not* the output of an XOR gate by sampling W_i^0 at random (as before) but then setting $W_i^1 = W_i^0 \oplus R$. For k an output wire of an XOR gate with input wires i, j, the evaluator sets $W_k^0 = W_i^0 \oplus W_j^0$ and $W_k^1 = W_k^0 \oplus R$. Note that if the circuit evaluator holds input-wire keys W_i, W_j associated with the input wires to an XOR gate, he can compute the corresponding key for the output wire k of that gate as $W_k = W_i \oplus W_j$. Thus, no garbled tables need to be prepared or sent for such gates.

We modify this technique to permit efficient parallel gate generation. In particular, note that when the circuit alternates between non-XOR gates and XOR gates in the 'encrypted' circuit, this creates a dependency amongst wire-labels that would require gates to be generated in leveled order. This is because the output wire labels from the first XOR gates on level $i-1$ need to be computed for use as the input wires on level i. This creates a dependency regardless of whether or not the gates on level i are XOR or truth tables.

Our modification operates by first virtually generating the labels (Sec. 7.3) for all wires in the circuit, even if the wire is the output

[5] There are two issues: i) wires in level $i+1$ of the circuit will now depend on the gates in level i, making parallel generation of the circuit difficult; and ii) during evaluation, about a 1/4 of the cores evaluating encrypted gates would evaluate to the missing row, and require different code than is required for the remaining 3/4 of the cores (causing warp divergence).

[6] John the Ripper is a brute force password hashing software suite.

wire from a XOR gate. This *differs* from the original Free-XOR technique which *does not randomly generate labels for XOR gate output wires*. Then, after wire label generation, during the generation of XOR gate i, we calculate a label offset (V_i) and a p-bit offset (P_i) that is unique for XOR gate i in the circuit. In the original Free-XOR technique it is at this point where an XOR gate's output wire labels would be generated. Instead, we make use of V_i and P_i to modify our XOR gate to our previously generated wire label and p-bit. The V and P values for every XOR gate can be calculated in parallel on the GPU. Our scheme adds two bitwise XOR operations to XOR gate generation, but this increased overhead is minuscule compared to locating every XOR gate chain and then serially computing each gate in the chain. The calculation of our V and P values during generation are as follows:

1. For each XOR-Gate G_i with wires $W_c = \text{XOR}(W_a, W_b)$ where $W_a = [\langle k_a^0, p_a^0\rangle, \langle k_a^1, p_a^1\rangle]$, $W_b = [\langle k_b^0, p_b^0\rangle, \langle k_b^1, p_b^1\rangle]$, $W_c = [\langle k_c^0, p_c^0\rangle, \langle k_c^1, p_c^1\rangle]$, and tuple $[V_c, P_c]$:
   (a) Set value $V_c = k_a^0 \bigoplus k_b^0 \bigoplus k_c^0$ for wire W_C
   (b) Set value $P_c = p_a^0 \bigoplus p_b^0 \bigoplus p_c^0$ for wire W_C

The use of V and P during evaluation are as follows:

1. For each XOR-Gate G_i with wires $W_c = \text{XOR}(W_a, W_b)$ where $W_a = \langle k_a, p_a\rangle$, $W_b = \langle k_b, p_b\rangle$, and tuple $[V_c, P_c]$:
   (a) Compute garbled output value $W_C = \langle k_c, p_c\rangle$ which is equal to $\langle k_a \bigoplus k_b \bigoplus V_c, p_a \bigoplus p_b \bigoplus P_c\rangle$

Communication Costs: As discussed, this modification increases the communication costs by adding an extra 'key' for each XOR gate in exchange for significant parallelized speed improvements. This modification is also incompatible with the Garbled Row Reduction (GRR) optimization [10], which performs more computation in exchange for less communication because GRR also induces a dependence on a circuit's wire labels.

7.3 GPU Wire Creation and Gate Generation

Our system implements garbled circuit generation on the GPU in parallel. We first note a method of virtually generating all label pairs and permutation bits $\langle k_a^0, p_a^0\rangle, \langle k_b^1, p_b^1\rangle$ for every wire in the circuit. Because of the memory hierarchy, one does not wish to generate all of the keys associated with labels initially, as the costs of moving and storing these keys in memory is substantial. Instead, we use wire indexes from the circuit description (which are much smaller than cryptographic keys), as inputs to a pseudo-random function generator (PRFG), which outputs the labels for a given wire. Specifically, we output $k_a^0 = F_s(a)$ and $k_a^1 = F_s(a) \oplus R$ for a PRFG F with a circuit specific seed s and the global Free-XOR offset R. The permutation bits are handled similarly. Note that wire-labels are not actually precomputed, but rather only virtually assigned, and computed when constructing the gates that are attached to a given label, for implementation optimization reasons.

7.4 GPU Evaluation

The entire circuit cannot be evaluated in parallel on the GPU. The gates in the circuit must be topologically sorted, and then evaluated. That is, the circuit must be broken into smaller sub circuits such that each subcircuit S has a start gate level G_s and end gate level G_e.

Our API starts GPU evaluation by iteratively transferring several consecutive levels of truth tables and XOR gates to the GPU's memory, and then evaluating them. The next grouping of levels can be asynchronously transferred to the GPU while the previous grouping of levels is being evaluated. At each level a separate kernel must be used to evaluate XOR and truth table gates.

CPU Evaluation.

Besides GPU Evaluation of garbled circuits we also implemented system to evaluate garbled circuits on the host both serially and in parallel. Our parallel CPU implementation makes use of OpenMP threads (cf. Sec. 5). The CPU parallel and serial evaluation work using the similar process as GPU evaluation, but importantly CPU evaluation does not need to perform any data transfers. The CPU evaluation algorithm iterates over each circuit level in the same manner as GPU evaluation. Serial CPU evaluation will iterate serially over each gate in a level and evaluate it. The parallel CPU evaluation algorithm iterates over gates in a level *in parallel*, where they are divided up equally among all available threads on the machine. Thus if a level contains N gates and the machine has t threads, each thread will process $\frac{N}{t}$ gates. No balancing is done to try and ensure a consistent mix of XOR and truth table gates, as the overheard was deemed higher than the benefit. Given CPUs are MIMD processors there is no need to worry about divergent branches, thus parallel evaluation on the CPU is a more straightforward process.

8. RESULTS

In this section, we present several experiments that support the main claims of this paper and give data on GPU circuit generation, GPU evaluation, and multi-core CPU evaluation. Given that there are now several implementations in different security models, such as HbC, 1BM, and malicious security, it is clear a critical metric to the performance of these systems is the number of gates one can generate and evaluate per core per second. In the HBC and 1BM security model, gate generation and evaluation are key, as Huang et al.[9] also note. Frederiksen and Nielsen suggest that communication is the fundamental limiting factor in the malicious model, but we recall that there are are now several communications improvements that will help to alleviate communication overhead, as discussed in Section 6, which suggest that these metrics should still be of some concern in the malicious model. We do not address communication or its latency here, as it is not in scope of our investigation on the use of different parallelizing technologies to implement efficient and practical circuit generation and evaluation. We discuss this in the Future Work section.

We show through experiment that circuit generation in the HbC and 1BM security models can dramatically benefit from a combination of the fine-grained parallelization that has not been exploited in prior works and our modification of the Free-XOR technique. Further, it can easily be accommodated on SIMD-style architectures such as GPUs. This applies to creating individual circuits, and can be carried forward to many duplicates for cut-and-choose scenarios, although the extra communications costs, and the availability of other parallelization techniques in that model may make our approach for that security model less feasible: more experiments need to be performed.

Finally, we show that circuit evaluation is more difficult to parallelize for individual circuits, but can perform better in the cut-and-choose scenario of malicious security.

8.1 Explanation of our Experiments and Data

Producing fair comparisons between different garbled circuit systems is currently challenging. In a perfect world we would execute all systems on the same machine using the same circuit descriptor files, and provide results. Unfortunately, we do not have access to all of the systems and circuit description files necessary to do this. We were able to get the Frederiksen and Nielsen [5] system work-

ing on two of our GPU systems for a direct comparison of our performance to theirs. We do not have interchangeable circuit formats, but we can provide comparisons on a per gate basis. Interestingly, their system performs better on an older architectural generation of GPU card then it was designed to function, so we compare their best performance and ours on both generations of cards. Further, since their implementation is in the malicious model, we cannot simply compare execution times of their many cut-and-choose generations and evaluations with a single generation or evaluation on our system. We took different AES circuits and "copy and pasted" multiple independent copies into one file to simulate the workload needed to generate or evaluate many copies of the circuit in the cut-and-choose protocol, and then compare on a per gate basis.

In the case of Kreuter at al. [15], we have recently been able to support generating and evaluating circuits form their most recent compiler [14], allowing for comparison of generation between the two systems on identical circuits, but we have not yet been able to fully integrate their system with our pipelining code, so we can only compile those circuits for which the entire final circuit fits on the GPU at one time. Larger circuits that need to be broken up and pipelined onto the GPU cannot yet be directly compared. For this reason, the comparisons of our system halt in the experiments when circuit sizes reach the maximal that will fit on the graphics cards. We note that of our two systems, one system's card has a newer architecture than the other, and so can support slightly larger circuits. Unfortunately, we did not have access to a version of Kreuter et al.'s system that would work on a non-clustered machine, so we could not provide bare-metal side by side comparisons. Therefore, in the case of Kreuter et al. [14], we take the results from their paper and compare them with the same circuits on our machines. All reported results from our experiments are the average of 100 runs. Experiments from Kreuter at al. [14] report average results from 50 runs.

Most prior work in the area benchmarks the time it takes to generate and evaluate various circuits. This process indirectly benchmarks the number of gates generated or evaluated per second. However, this is often run on systems with varying numbers of cores, and to a lesser extent varying speeds. We report results on the average number of gates generated or evaluated per second per core. We note this metric seems relatively stable, and thus we use it for a near apples-to-apples comparison. Table 1 has details for the comparison systems. We note that even though EC2 has multiple GPUs, only one is used in the results presented.[7] EC2 is run on Amazon's elastic compute infrastructure, and is running under a Xen hypervisor. Since we do not have direct access to the bare metal, we cannot determine how much overhead the Xen hypervisor entails, but Xen project benchmarks suggest, assuming appropriate kernel patches have been applied, a 0-30% performance decrease [2].

8.2 GPU Circuit Generation

We ran circuit generation on the EC2 and Tie systems (cf. Table 1). We first compare our results to those of Frederiksen and Nielsen [5] in Fig. 1a. We remind the reader that we compare their circuit generation times from experiments where they have similar, but not identical circuits, due to the need to simulate the cut-and-choose malicious protocol, and further, while we did have access to their circuit file, we could not execute it directly as we do not support their file description language in our system, and their binary file format was not conducive to easy translation. Thus, we show in Fig. 1a that under similar workloads our scheme outperforms theirs on the same hardware using the metric of gates generated per second. Observe that we generate gates at about 2.3 times the rate on the Tie system compared to Frederiksen and Nielsen on the EC2 system. Observe that we generate gates at about 3 times the rate on the Tie system compared to Frederiksen and Nielsen. This is the benchmark system, as Frederiksen' and Nielsen's code is targeted at compute capability 3.X CUDA cards.

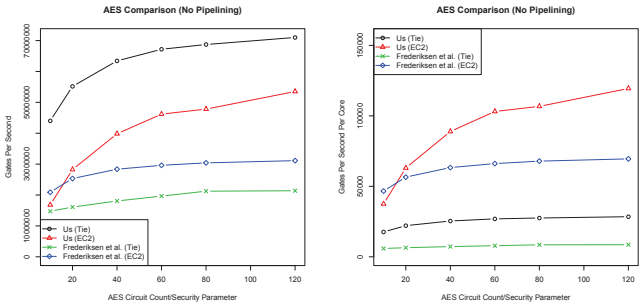
[7]We discuss multiple GPUs in Sec. 9 with respect to future work

System	CPU	Core/ Thrd.	GHz	Ram (GB)	GPU
Kreuter et al. [15]	Xenon E5506	4	2.13	8	N/A
EC2	Xenon X5570	8/16	2.93	24	Tesla S2050
Tie	Xenon E5-2620	12/12	2	64	Tesla K20

GPU	Cores	SMs	GHz	Memory (GB)	Compute Capability
S2050 (EC2)	448	14	1.15	2.7	2.0
K20 (Tie)	2496	13	0.71	4.8	3.5

Table 1: Benchmark system descriptions. EC2 runs a Xen virtual machine.

As the number of cores on systems can be highly variable, in Fig. 1b we calculate the average rate of gate generation per core for the two systems, to help with understanding performance on other GPU cards with varying numbers of cores. Note that in the benchmarks reported in Figs. 1a and 1b we have commented out any code in our system necessary to split large circuits into smaller sub-circuits so that they can fit onto the GPU, as Frederiksen and Nielsen have no such corresponding code as they simply assume the circuit will fit. Thus we are not penalized for computing overhead that the other system also does not compute.

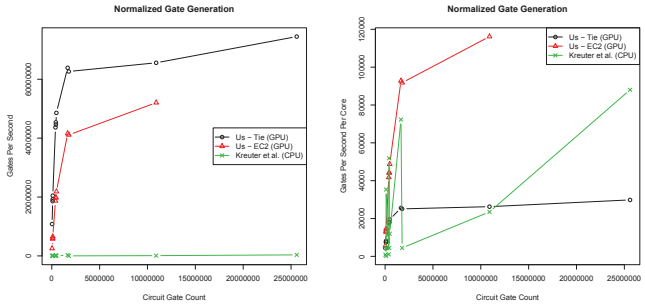


(a) (b)

Figure 1: 1a) Circuit gate generation rates of [5] vs. our technique using fully parallelizable circuit generation. 1b) Gates generation rate per multi-processor on differing circuit sizes.

Next, we considered a number of different circuit sizes from both Kreuter et al.[14], and circuits that we have constructed. Given our support of PCF we can compare the same circuits as are tested by Kreuter et al.[14]. We see in Fig. 2, the absolute performance of our system versus that of Kreuter et al. in terms of Gates per sec, and then in Figs. 4a and 4b the relative performance per core. Note that performance per core is relatively stable across medium-to-large circuit sizes. Recall that our cores are substantially more abundant, and have lower cost and energy usage that those of Kreuter et al.

Using the metric of gates per second we find our system, in the case of generation, provides significantly higher generation rates: approximately three orders of magnitude. Our system tops out at around 75 million gates per second, while Kreuter et al tops out at 0.35 million gates per second. We note that their system is built for cluster computing, and so they pay a significant overhead to support it.



(a) Gates Gen vs. Time (b) Gate Gen Per Core Per Sec vs. Gate Count

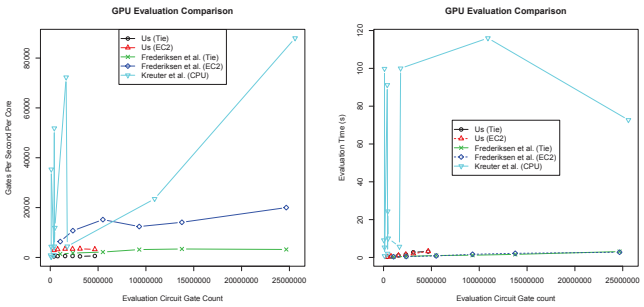
Figure 2: Gate Generation Times comparing to Kreuter et al.[14].

8.3 GPU Evaluation

While on generation we significantly outperform other systems, we only comparable performance to the CPU evaluation techniques of Kreuter et al. [15], and are slightly less efficient on a than the current implementation by Frederiksen and Nielsen [5]. Results are given in Fig. 3.

The advantage that the cut-and-choose protocol entails to parallel evaluation, especially on the SIMD architecture, makes it difficult for an HbC or 1BM model protocol to remain competitive. Our evaluation problems seem to stem from two factors: i) It is difficult to keep the GPU fully engaged in processing, due to the limited width of any level of a circuit (recall level i of a circuit must be evaluated before level $i + 1$); and, ii) The lack of memory coalescence in our circuit evaluation data structure seems to impose harsh time penalties on our circuit evaluation times, due to poor poor memory read/write performance. Memory coalescence occurs on a GPU when all the threads in a warp access adjacent memory locations. Problem ii) is one we believe we can partially improve upon in future work, although we doubt it is possible to achieve the same levels as the cut-and-choose protocol permits (discussed below). Problem i) is inherently more problematic for the HbC and 1BM security model protocols, as one can never have guarantees that there are k identical copies of each gate to evaluate, nor do we have the ability to naturally multiply the width of circuits by a factor of $\mathcal{O}(k)$. For naturally large circuits, there may be some hope.

Recall core utilization rates and memory coalescence are less of an issue for Frederiksen and Nielsen: not only are they in fact computing many copies of the AES circuit in the malicious model as we are, but their evaluation algorithm is guaranteed of this fact. This allows them several advantages when constructing kernels to evaluate their circuits. In particular, they can solve the two problems above. First, they can construct a kernel for evaluating each gate in a circuit, and they can evaluate gates from lowest level to the highest. As long as these kernels are scheduled in a leveled order—something easily done— the GPU need never sit with low usage while waiting on kernels to complete a level. Second, since



(a) Comparing our GPU Eval Per Sec Per Core (b) Comparing our GPU Eval Overall

Figure 3: GPU Evaluation Times with comparison to Kreuter et al. [14], Frederiksen and Nielsen [5] and our GPU implementation.

the evaluation is guaranteed that it is executing multiple copies of an identical circuit, it is easier to setup kernels that i) avoid warp divergence, as warps will never process different gate types, and ii)coalesce circuit data in the GPU's global memory, by simply storing each circuits data adjacent in memory. Note that both of these solutions depend on the GPU taking advantage of multiple identical copies of the same circuit executing.

We see that our GPU marginally outperforms Kreuter et al., suggesting that they are paying a heavy price for using MPI on a single machine (but of course, they are designed to run on large compute clusters, and huge circuits where such performance penalties should be amortized).

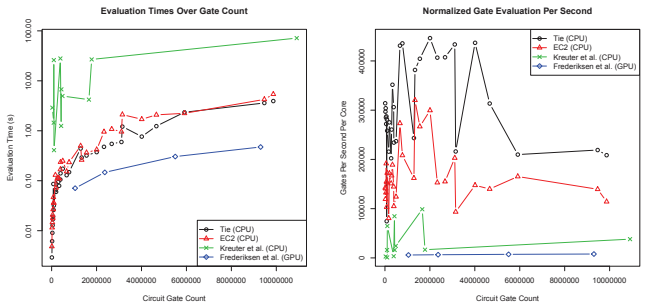
8.4 CPU Evaluation

Due in part to the seemingly structural problems of evaluation on a SIMD GPU, we implemented a multi-threaded CPU evaluation scheme in OpenMP. Results can be seen in Fig. 4. It is clear that a MIMD architecture, such as a multi-core CPU will not suffer from warp divergence or memory coalescing problems given their advanced memory controllers and internal logic. A lack of warp divergence removes the fear that large numbers of cores sit idle while a level is completed is less of a problem. Also, we do not need to create multiple distinct 'kernels' for different gate types, nor worry that different cores are evaluating different gates. Similarly, the fraction of cores that go unused while waiting for a level to complete, as a total fraction of compute power will be smaller.

While we continue to under perform Frederiksen and Nielsen, we improve over Kreuter et al, and show that their system is likely to benefit from the inclusion of threading within their nodes on the compute-cluster, as opposed to having all of the parallelism at the node level.

9. CONCLUSION, LESSONS LEARNED, AND FUTURE WORK

Given the ability of the GPU to generate large circuits (or large numbers of circuits) efficiently, and the CPUs better performance in evaluation, it seems that an implementation that aims to implement a cut-and-choose protocol, should do verification and generation on the GPU, and evaluation on the CPU in parallel. Similarly, 1BM implementations should implement generation on the GPU, and evaluation on the CPU. With appropriate pipelining these would be done in parallel. The technique introduced which allows



(a) CPU System Gate Count Eval vs. Time (b) CPU System Gate Eval per sec per core vs. Gate Count

Figure 4: Our Evaluation Times as implemented on the CPU with comparison to Kreuter et al. [14] and Frederiksen and Nielsen [5] .

XOR gates to be generated in parallel greatly helps in the circuit gate generation rate.

While we do not report the results here, we have initial work showing there is potential for multiple GPUs to be used in a single system to further speed generation and evaluation results, but a more careful implementation must be done that carefully splits work amongst the GPUs, and takes into consideration the single-bus bottleneck, or card-to-card memory transfer. We plan to pursue these directions as future work.

It is clear that in order for better performance comparisons to be made in the future, there needs to be a test-bank of standard circuits designed. They must be delineated in a standard file format that all future implementations can parse (although, they may further process in this format). Currently, each implementation in the field is rolling its own file format. The recent development of `MPCLounge` aims to keep track of such circuits. Similarly, the SCAPI project by Ejgenberg et al. will help in providing a long term supported test environment [4].

10. ACKNOWLEDGEMENTS

The authors would like to thank the NSF and DARPA for funding, Jonathan Katz for discussion and aid on preliminary work, and Tore Frederiksen and Ben Kreuter for aid with their systems. This work was supported by an AWS in Education grant.

11. REFERENCES

[1] Bellare, M., Hoang, V. T., Keelveedhi, S., and Rogaway, P. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy* (2013), IEEE Computer Society, pp. 478–492.

[2] Campbell, I. Baremetal vs. xen vs. kvm — redux. `http://blog.xen.org/index.php/2011/11/29/baremetal-vs-xen-vs-kvm-redux/` (Nov 2011).

[3] Damgård, I., Pastro, V., Smart, N. P., and Zakarias, S. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO* (2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*, Springer, pp. 643–662.

[4] Ejgenberg, Y., Farbstein, M., Levy, M., and Lindell, Y. Scapi: The secure computation application programming interface. *IACR Cryptology ePrint Archive 2012* (2012), 629.

[5] Frederiksen, T. K., and Nielsen, J. B. Fast and maliciously secure two-party computation using the gpu. Tech. rep., Cryptology ePrint Archive, Report 2013/046, 2013. http://eprint. iacr. org, 2012.

[6] Goldreich, O. *Foundations of Cryptography, vol. 2: Basic Applications*. Cambridge University Press, Cambridge, UK, 2004.

[7] Goyal, V., Mohassel, P., and Smith, A. Efficient two party and multi party computation against covert adversaries. In *EUROCRYPT* (2008), N. P. Smart, Ed., vol. 4965 of *Lecture Notes in Computer Science*, Springer, pp. 289–306.

[8] Huang, Y., Evans, D., Katz, J., and Malka, L. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium* (2011).

[9] Huang, Y., Katz, J., and Evans, D. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 272–284.

[10] Huang, Y., Shen, C.-h., Evans, D., Katz, J., and Shelat, A. Efficient secure computation with garbled circuits. In *Information Systems Security*. Springer, 2011, pp. 28–48.

[11] Ishai, Y., Kilian, J., Nissim, K., and Petrank, E. Extending oblivious transfers efficiently. *Advances in Cryptology-CRYPTO 2003* (2003), 145–161.

[12] Jang, K., Han, S., Han, S., Moon, S., and Park, K. Sslshader: cheap ssl acceleration with commodity processors. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation* (2011), USENIX Association, pp. 1–1.

[13] Kolesnikov, V., and Schneider, T. Improved garbled circuit: Free XOR gates and applications. pp. 486–498.

[14] Kreuter, B., Mood, B., shelat, a., and Butler, K. Pcf: A portable circuit format for scalable two-party secure computation. In *To Appear in USENIX Security 2013* (2013).

[15] Kreuter, B., Shelat, A., and Shen, C.-H. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21st USENIX conference on Security symposium, Security* (2012), vol. 12, pp. 14–14.

[16] Lindell, Y. Fast cut-and-choose based protocols for malicious and covert adversaries. In *CRYPTO (2)* (2013), R. Canetti and J. A. Garay, Eds., vol. 8043 of *Lecture Notes in Computer Science*, Springer, pp. 1–17.

[17] Lindell, Y., and Pinkas, B. A proof of security of Yao's protocol for two-party computation. 161–188.

[18] Malkhi, D., Nisan, N., Pinkas, B., and Sella, Y. Fairplay—a secure two-party computation system. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13* (2004), USENIX Association, pp. 20–20.

[19] Mohassel, P., and Franklin, M. K. Efficiency tradeoffs for malicious two-party computation. In *Public Key Cryptography* (2006), M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., vol. 3958 of *Lecture Notes in Computer Science*, Springer, pp. 458–473.

[20] Naor, M., and Pinkas, B. Computationally secure oblivious transfer. *Journal of Cryptology 18*, 1 (2005), 1–35.

[21] Pinkas, B., Schneider, T., Smart, N., and Williams, S. Secure two-party computation is practical. pp. 250–267.

[22] Pinkas, B., Schneider, T., Smart, N. P., and Williams, S. C. Secure two-party computation is practical. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology* (Berlin, Heidelberg, 2009), ASIACRYPT '09, Springer-Verlag, pp. 250–267.

[23] Pu, S., Duan, P., and Liu, J.-C. Fastplay–a parallelization model and implementation of smc on cuda based gpu cluster architecture. Tech. rep., Cryptology ePrint Archive, Report 2011/097, 2011. http://eprint. iacr. org, 2011.

[24] Shelat, A., and Shen, C.-H. Two-output secure computation with malicious adversaries. In *EUROCRYPT* (2011), K. G. Paterson, Ed., vol. 6632 of *Lecture Notes in Computer Science*, Springer, pp. 386–405.

[25] Yao, A. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on* (1986), IEEE, pp. 162–167.