Computing Privacy-Preserving Edit Distance and Smith-Waterman Problems on the GPU Architecture

Track: Cryptographic Technologies, Secure Compututation

Shi Pu
Department of Computer Science and Engineering
Texas A&M University
College Station, TX 77840
shipu@cse.tamu.edu

Jyh-Charn Liu
Department of Computer Science and Engineering
Texas A&M University
College Station, TX 77840
liu@cse.tamu.edu

***Abstract*—This paper presents privacy-preserving, parallel computing algorithms on a graphic processing unit (GPU) architecture to solve the Edit-Distance (ED) and the Smith-Waterman (SW) problems. The ED and SW problems are formulated into dynamic programming (DP) computing problems, which are solved using the *Secure Function Evaluation* (SFE) to meet privacy protection requirements, based on the semi-honest security model. Major parallelization techniques include mapping of variables to support collision-free parallel memory access, scheduling and mapping of gate garblers on GPU devices to maximize GPU device utilization, and latency minimization of context switch for computing steps in the DP matrix. A pipelined GPU-CPU interface is developed to mask latency of CPU housekeeping components.**

The new solutions were tested on a Xeon E5504 at 2GHz plus a GTX-680 GPU (as *generator*), connecting an i7-3770K at 3.5GHz plus a GTX-680 GPU (as *evaluator*) via local Internet. A 5000×5000 8-bit alphabet ED problem requires roughly 1.88 billion non-free gates, and the running time of around 26 minutes (roughly 1.209×10^6 gate/second). A 60×60 SW problem is computed in around 16.79 seconds. Compared to the state of art performance [5], we achieved the acceleration factor of 12.5× for the ED problem, and 24.7× for the SW problem.

Keywords*—*Secure Function Evaluation, dynamic programming, GPU, acceleration

I. INTRODUCTION

The two-party Secure Function Evaluation (SFE) [13, 14] based *dynamic programming* scheme has been proposed for privacy-preserving matching of genomic data pairs. In SFE, two players jointly compute an arbitrary logic function *f(x,y)* while they keep their multi-bit inputs *x* and *y* private at all time. Yao's *garbled circuit*s (GC) [13] and the *oblivious transfer* (OT) protocol [19] have been widely adopted for SFE implementation. One player assumes the role of a *generator*, who constructs the garbled circuit for *f*. The other player assumes the role of *evaluator*, who evaluates the circuit. To protect data privacy, each input bit (plain-text version) is represented as a pair of *wire label*s (tens of bits), and each Boolean logic is transformed to a sequence of a garbling (encryption) step on the generator side, plus a de-garbling (decryption) step on the evaluator side.

SFE has been used as the building block for privacy-preserving applications such as secret auctions [15, 25], biometric or genomic computation [3, 4, 5], facial recognition [2, 7, 16, 17] and encryption [2, 5, 6]. To enable broader adoption of SFE for real world applications, in this paper we focus on acceleration of the SFE based *dynamic programming* (DP) [33]. We presented new parallel algorithms to solve the privacy-preserving ED and SW problems. In sharp contrast to existing SFE functions like AES, Hamming distance, RSA, or Dot product, DP needs complicated computations (addition, min/max, lookup tables, subtract etc.), and its computing steps exhibit strong interdependency.

In the state of art solutions [4, 5], it took 3.5 hours and 7 minutes respectively to solve a privacy-preserving ED problem (2000×10000 8-bit alphabet), and a 60×60 SW problem. In this paper, we achieve speedups of 12.5x (ED problem) and 24.7x (SW problem) through optimization of GPU resource management, efficient DP computing process mapping, and tight integration of GPU and CPU interactions.

Details of our design will be discussed later. Succinctly put, our contribution includes: (1) a high-throughput gate garbler/de-garbler fully loading GTX-680; (2) for the ED problem and the SW problem respectively, the generator's/evaluator's GPU resource mapping policies that maximally utilize the gate garbler/de-garbler; (3) a pipelined CPU-GPU computing architecture to support end-to-end computing service. (4) We adopt several known optimizations: *free-XOR* [8], *oblivious transfer extension* [18], permute-and-encrypt [1], *efficient lookup-table design* and *compact circuit*s [5], and solve confliction between the free-XOR technique and our wire label pre-assigning scheme. Our GPU-based random wire label generator based on the mathematical model used in MIRACL [26].

A. Parallel Computing Model for GC

We assume that a GC is constructed from a number of 2-bit input, 1-bit output gate (G), c=G(a,b), where the one bit "*wire*" a (b) is provided by generator (evaluator). To compute G jointly, the generator produces *wire label*s $\{k_a^0,k_a^1\},\{k_b^0,k_b^1\}$, $\{k_c^0,k_c^1\}$ for each possible value of a, b, and c. Through the OT protocol [19], the generator sends encrypted $\{k_b^y\ k_b^{y'}\}$ to the evaluator, but the evaluator can only decrypt k_b^y that matches y, the actual value of b. Based on a random oracle model proposed in [18, 20], a virtually unlimited number of OT computations can be encoded into 80 1-out-of-2 OT transactions [18, 20], where 80 is a security parameter. The 80 1-out-of-2 OT transactions can be computed in 0.6s [5]. That is, the primary computing bottleneck is garbling/de-garbling the vast number of circuits, not the OT, for the SFE.

The computing logic of G is a four-entry truth table T{T_{00}, T_{01}, T_{10}, T_{11}}, where each entry of value 0(1) is associate with the output wire label $k_c^0(k_c^1)$. The garbled truth table is a random permutation of the four cipher-texts: $\{Ek_a^0(Ek_b^0(k_c^{T00}))$, $Ek_a^0(Ek_b^1(k_c^{T01}))$, $Ek_a^1(Ek_b^0(k_c^{T10}))$, $Ek_a^1(Ek_b^1(k_c^{T11}))\}$, here E denotes the encryptor (also known as garbler). In the end, the evaluator learns the wire label k_c^{Txy} that represents the real value of G given inputs a=x and b=y, while he does not know x. Using G as building blocks, we use SHDL in Fairplay [1] to construct GCs that can solve the privacy-preserving ED problem and SW problem.

Our parallelization strategy is loosely divided into the *GC level* and *DP level*. In the GC level, the vast number of gates are concurrently garbled by the generator and de-garbled by the evaluator. On top of the GC level, the DP level computes the DP matrix of the ED or SW problem. The GC level is designed to meet the *ultra-short* security in TASTY [2], and the DP level aims to maximize processor utilization while enforcing the interdependency relationship between computing steps.

The DP level design aims to maximize resource utilization. As shown in Fig. 1, the N×N DP matrix is processed into 2N-1 *slices*, W= {S_1, S_2, …, S_{2N-1}}. Fig. 1 shows that *GC-slots* (entries in the DP matrix) on the same slice are independent, and thus can be concurrently garbled (de-garbled) on the generator (evaluator) side. The degree of parallelism increases from S_1, S_2 until S_N, and then it decreases from S_N to S_{2N-1}. For garbling GC-slots on S_i, wire labels of GC-slots' outputs on slices S_{i-2} and S_{i-1} are re-used, thus multiple slices can be garbled simultaneously once wire labels of all GC-slots' outputs are pre-assigned. This observation is implemented as a

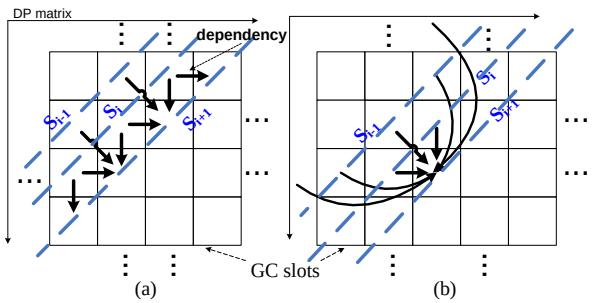

Fig. 1. Parallel Computing Models for (a) the ED problem, and (b) the SW problem.

cross-slice mapping policy (CSMP). However, for de-garbling GC-slots on S_i, the pre-requisite is the de-garbled outputs on slices S_{i-2} and S_{i-1}. So, the de-garbling process can only de-garble one slice at a time. As a result, garbling the N×N matrix is transformed to a 1-D vector which is mapped to GPU units. For de-garbling, S_i is mapped to GPU units after S_{i-1} is completed.

The rest of the paper is organized as follows. Section 2 discusses the GPU (de-)garblers. Sections 3 & 4 present designs in the DP level for the ED problem and SW problem respectively. The SFE system is presented in section 3. Section 5 gives the related work. Section 6 concludes the paper.

II. GPU-Based Gate (De-)Garbler

Our gate garbler is implemented on the CUDA (Compute Unified Device Architecture). GTX-680 is a GK104 generation device [10, 34], which contains 8 *streaming multiprocessors* (SMX). Although each SMX supports parallel processing of 32 threads, called a *warp*, per clock, it usually simultaneously runs multiple warps of threads for better utilization of its pipeline. Each SMX has 64K 32-bit registers and 64KB on-chip *shared memory*/L1 cache, which is organized into 32 64-bit banks. 8 SMXs share 2GB 256-bit wide slow *global memory*. A program on GPU is called a *kernel function*. Its input setup, parallelism configuration, launching and output read-back are controlled by a host thread on CPU. At runtime, following the *Single Instruction Multi Threads* (SIMT) architecture, each GPU thread runs one instance of the kernel function.

For garbling, an arbitrary entry T_{xy} in the truth table of a gate G, is garbled (encrypted) as $Enc_{x,y}(k_c^z)$ = $H(k_a^x \| k_b^y)$ XOR k_c^z, where H is the garbling function, k_a^x, k_b^y and k_c^z are wire labels, "||" is concatenation. Like Huang et al. [5], we adopt 80-bit as the length of wire labels so that the ultra-short security in TASTY [2] is achieved. Candidates of H are SHA-1 [5], AES-256 [6] supported by AES-NI, SHA-256, or other cryptographic hash functions. Among these choices, we adopted SHA-256 as H because it has similar computing cost as SHA-1 [29, 30], but without its vulnerability [27]. AES-256 is not chosen because it is 3 times slower than SHA-1 on GPU [9]. So that, $Enc_{x,y}$ (k_c^z) = SHA-256(k_a^x || k_b^y || i) XOR k_c^z, where i is a 32-bit unique gate index in a garbled circuit. This SHA-256's input is a 192-bit block, and output a 256-bit

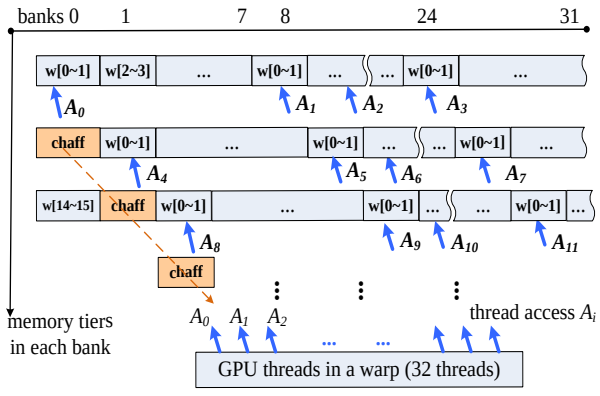

Fig. 2. Conflict free parallel access to share memory banks.

digest. The de-garbling (decryption) is Dec($Enc_{x,y}$ (k_c^z)) = SHA-256(k_a^e || k_b^e || i) XOR $Enc_{x,y}$ (k_c^z), where k_a^e, k_b^e are wire labels obtained from OT or previous de-garbling.

We adopted the code base in PolarSSL [32] for the SHA-256 implementation, with the following adjustments. Here, one round of SHA-256 is divided into four steps, each of which produces 16 (32-bit) words W[0~15] based on the elements in W computed in the current and previous steps. At end of each step, W[0~15] are used to update the 8 32-bit digest. A total of 40 (32-bit) words of space are needed for each round. That is, 16 (32-bit) registers to store W[0~15] in the current step, 8 (32-bit) registers to store the digest, and one block of 16×32-bit shared memory is assigned to each thread to store W[0~15] produced in the past step, the storage format of such a shared memory block per thread is illustrated in Fig. 2. Here, each block resides on eight shared memory banks.

GPU threads in a same warp will be stalled when some of them attempt to access different *tiers* (a low level GPU architecture) on the same memory bank. To eliminate the often hidden shared memory access conflicts, we fill in a strip of 64-bit *chaff* spacers, one in the front of every fourth-thread's W[0~15]. This way, parallel memory accesses {A_i, A_{i+4}, A_{i+8},…A_{i+28}} issued by threads i, $i+4$, $i+8$, …$i+28$ (i=0,1,2, or 3) to read W[0~15] of the same index in its own W array will access distinct memory banks with no conflict. Fig. 2 illustrates the case i=0 and {A_0, A_4, A_8… A_{28}} read W[0].

Each SHA-256 gate garbler thread uses 57 registers, where GK104 allows up to 63 registers per thread. Each SMX has 640 threads (20 warps), its 64KB on-chip memory is organized as 48KB of shared memory, plus 16KB of L1 cache, and 41.25KB of the 48KB shared memory is occupied by W[0~15] with chaff spacers. The degree of parallelism is 5120=8 SMX × 640 threads. The latency is 304ms for a GPU thread to read in a block of 192 bits 10000 times. The throughput is roughly 30.27Gbps, taking the GPU-CPU data exchange time into account, which is similar to the result of SHA1 on GTX-580 [28]. Intel reported their SHA256 achieved 11.5 cycles/byte (equivalent to 2.47Gbps) on a single core of Intel i7 2600 in 2012 [36]. Next, we discuss our designs in the DP level.

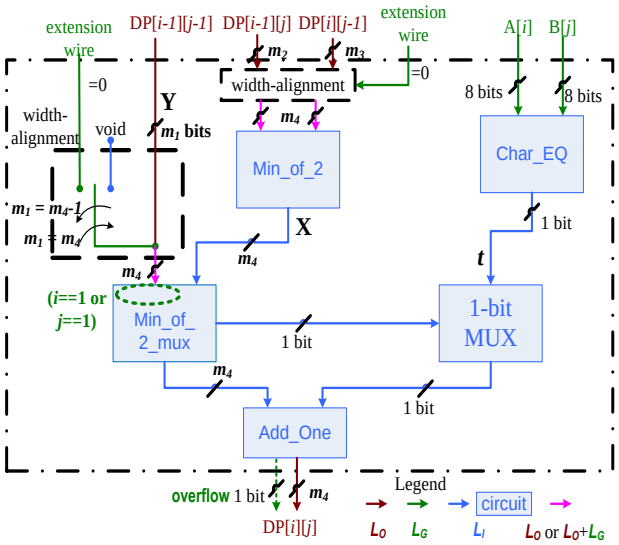

Fig. 3. The SFE building block (a GC-slot at *DP[i][j]*) for the ED problem.

III. Privacy-preserving Edit Distance Computing

A. SFE Building Block for Edit Distance

Given two input strings A[N] and B[N] from the two players respectively, solving the ED problem is essentially processing of an (N+1)×(N+1) DP matrix, in which a GC-slot DP[i][j] (i,j∈[0,N]) is computed as: DP[i][0] = i, DP[0][j] = j; and if i,j∈[1,N], DP[i][j] = (Y > X) ? (X+1) : (Y+t), where t = (A[i] ≠ B[j]), X=min(DP[i-1][j] , DP[i][j-1]), and Y=DP[i-1][j-1] [5]. Referring to Fig. 3, one GC-slot includes two *Min_of_2* circuits (*Min_of_2* and *Min_of_2_mux*), one *Char_EQ* circuit (compute t), and one *Add_One* circuit.

The maximum value of an arbitrary GC-slot DP[i][j] is max(i, j). To minimize the gate count, it is highly desirable to implement GC-slots based on the actual number of bits required for inputs in the multi-version compact circuits [5]. We adopt the multi-version *Min_of_2*, *Min_of_2_mux* and *Add_One* circuits in [5], so that one of the multiple versions is activated for a GC-slot according to its inputs' bit widths. The compact circuit [5] did not explicitly discuss how to handle variable bit widths. To solve this problem, we propose a width alignment scheme based on two 1-bit extension wires (see Fig. 3) for {DP[i-1][j], DP[i][j-1]}, and {X, Y}. The maximum possible values of inputs and intermediate results in a GC slot are listed in Table I. For {DP[i-1][j], DP[i][j-1]}, m_3=m_2 – 1(m_2=m_3 – 1) and the extension wire is activated for DP[i][j-1] (DP[i-1][j]) when i<j (i>j), and j (i) equals power of 2. Similarly, for {X, Y}, m_1= m_4 – 1 and the extension wire is activated for Y when i==j, and i is power of 2.

TABLE I. Maximum Possible Values of Inputs and Intermediate Results in a GC-slot DP[i][j]

	DP[i][j-1](width=m_3)	**DP[i-1][j](width=m_2)**	***X*(width=m_4)**	***Y*(width=m_1)**
i<j	max(i,j-1)=j-1	max(i-1,j)=j	min(i-1,j)=j-1	max(i-1,j-1)=j-1
i==j	max(i,j-1)=i	max(i-1,j)=j	min(i,j) = i	max(i-1,j-1)=i-1
i>j	max(i,j-1)=i	max(i-1,j)=i-1	min(i,i-1) = i-1	max(i-1,j-1)=i-1

B. Resource mapping policies and house keeping

On the generator side, as discussed earlier, under CSMP, GC-slots perform garbling in parallel when their paired wire labels are pre-assigned. CSMP partitions the DP matrix into multiple *task*s, each of which aims to fill up 5120 GPU gate garbler threads to maximize the speedup factor. For example, task[0] contains 5120 GC-slots, which is from slice 1, 2, 3, … up to a fraction of slice 101. When GPU runs the 5120 GC slots in lock-step, each GPU thread garbles its corresponding GC-slot gate by gate for the entire GC-slot.

Three major cases need to be considered for pre-assignments of paired wire labels. Referring to Fig. 3, L_O represents the set of paired labels for wires of a GC-slot's outputs. L_I represents the set of paired labels for wires internal

to a GC-slot and not connected to other GC-slots. L_G represents miscellaneous types of paired labels, and they are treated as a *"global"* set for the SFE to simplify memory management. Classification of the three groups of wire labels is not only important to efficient use of the GPU memory space, but also critical to synchronous accesses of memories by parallel threads. L_O and L_I are pre-assigned at initialization of a new task, but L_G at initialization of the whole SFE system. L_O and L_I are overwritten if they are associated with an XOR gate's output [8] by a calculation result based on the XOR gate's inputs' wire labels. Even though some L_O and L_I need to be overwritten during execution, they are still pre-assigned to simplify the wire label generation function at negligible costs.

Overwriting of labels in L_O occurs before garbling of a task. Generally speaking, wire labels in L_I are overwritten during garbling because no other GC-slots depend to them. However, some wire labels in L_I needs to be overwritten before overwriting of L_O if the value of the latter is dependent on that of the former, as shown in the following example. Here, the output of an XOR gate G_1 is the input of another XOR gate G_2, and the output of G_2 is also the GC-slot's output. Labels associated with G_1's (G_2's) output is in $L_I(L_O)$. Overwriting of labels for G_1 needs to be done before overwriting of labels for G_2, before garbling of a task.

L_O and L_I for GC-slots in a task are sequentially stored GC-slot by GC-slot. At initialization of the SFE system, a parser parses different versions of garbled circuits and calculates the version no. of circuits for each GC-slot in the DP matrix. After the maximum memory size for storing labels in L_O and L_I in a task is calculated, GPU memory space is statically allocated for a task, which is released until completion of the SFE computation. Static memory allocation eliminates costly *cuda_malloc*() and *cuda_free*() operations during computing. A dedicated memory block is reserved at the SFE initialization to store a copy of L_O for GC-slots in the latest three slices of the current task, so that they can be re-used in next task.

Next, we discuss cases related to L_G. For GC slots that compute DP[*1*][*j*], or DP[*i*][*1*], i.e., the second row and second column of the DP matrix, their *Min_of_2_mux* circuit's input DP[*i-1*][*j-1*] is a real value, rather than wire labels from other GC slots. As such, some gates in their *Min_of_2_mux* circuit need not perform garbling because they only accept inputs from the generator. We treat these gates the same as the generator's inputs and directly assign paired wire labels for these gates' outputs. For the *Add_one* circuit, the maximum possible value of its input is *j-1* (*i-1*) if *i*<*j* (*i*>*j*). The maximum possible value of its output is *j* (*i*), which needs an overflow bit for its correct representation when *j* (*i*) is power of 2. The next case is for the extension wires mentioned earlier. The last case is labels associate with the generator's input A[*N*] and the evaluator's input B[*N*]. They are global because they need to be used by multiple GC-slots.

The outputs of the garbling process are encrypted truth table entries and permute-and-encrypt bits [1] of all GC-slots

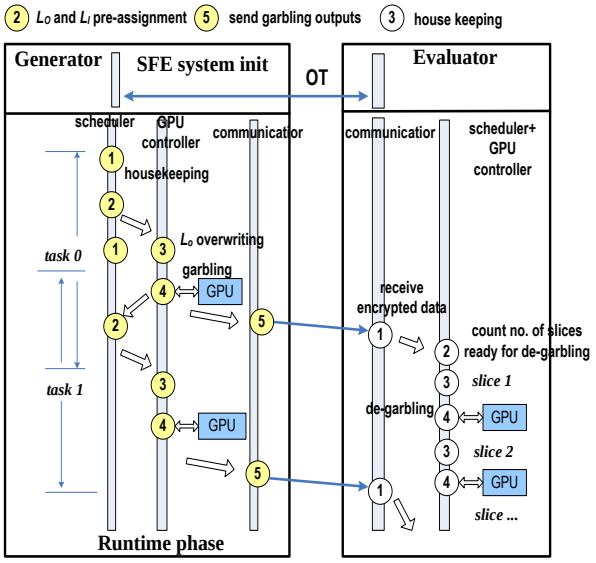

Fig. 4. The pipelined garbling & de-garbling process (ED)

in a task. Similar to L_O and L_I, memory of encrypted truth table entries and permute-and-encrypt bits in a task is also allocated statically. At completion of the garbling process of a task, the garbling outputs are copied from GPU to the host memory for network transfer to the evaluator. The actual utilization of the statically allocated memory space fluctuates with task execution. For instance, garbling output of task 0 in the reported test case needs less than 50% of the (statically allocated) memory space. As such, a compaction process is in place to identify and transmit only garbling outputs.

On the evaluator side, as discussed earlier, the de-garbling process runs in the order of slice-by-slice, because a GC-slot *DP[i][j]* can be de-garbled only after GC-slots DP[*i-1*][*j*], DP[*i*][*j-1*] and DP[*i-1*][*j-1*] in the previous two slices have been de-garbled. After receiving the garbling outputs of a task from the generator, the evaluator first calculates the number of slices ready for de-garbling. Each GC-slot in the slice is mapped to a GPU gate de-garbler thread. Taking the slice 101 as an example, the evaluator starts to de-garble it after receiving garbling outputs of task 1 since task 0 does not include all GC-slots in slice 101. Then the 101 GC-slots run on the GPU in lock-step, each GPU thread de-garbles its corresponding GC-slot gate by gate for the entire GC-slot.

The evaluator and generator have the same rules for classification of wire labels (as L_O, L_I, or L_G), but their management policies differ, except that on the evaluator only one of the two paired wire labels (on the generator) as de-garbling result. Also, on the generator, the assessment of maximum memory usage is based on one task, but on the evaluator, it is based on one slice. When the problem size *N* is small, the former requires much larger memory space than the latter. As *N* exceeds the level of physical parallelism supported by the GPU, the latter requires more memory space than the former.

Fig. 4 illustrates the pipelined processing flows between the garbling and de-garbling processes for the ED problem. The generator's step 1 (the evaluator's step 3) "house-keeping" calculates relative start addresses of L_O, L_I, L_G, and garbling output (garbled data as input, and de-garbling result) for each GC-slot in a task (slice). This step also sets up flags of extension wires and overflow bits for each GC-slot. The latency of housekeeping is masked by the pipeline.

TABLE II. PIPELINE EXECUTION TIME BREAK DOWN (ED)

Exec Time	Generator	Evaluator
SFE System initialization	6.92s	2.94s
Housekeeping	6.06s	23.04s
GPU garbling & de-garbling (without GPU-CPU data copy)	1062.95s (0.218 s/task)	136.55s (0.014 s/slice)
GPU-CPU data copy, resource mgnt	99.13s	50.21s
Total computing latency	1520s	345.3s

C. Experimental results

The test case is one 5000×5000 8-bit alphabet ED problem, which roughly costs 1.88 billion non-free gates. The generator runs 4883 tasks, and the evaluator runs 9999 slices. Table II lists the break down of execution times for major steps for the test case. The total computing latencies (1520s, 345.3s) do not include networking transmission latencies, nor the system initialization time. There exists a difference between the total computing latencies (row 5), and the sum of rows 2, 3, and 4. Such a difference is mainly spent on compaction of the garbling outputs. And on the evaluator side, the time difference is spent on a reverse process of the generator's compaction process, which normalizes lengths of the garbling outputs.

For the tested case study, the generator usually completes its total computing tasks when the evaluator completes 93% of de-garbling slices. The overall running time, excluding networking delays, to compute the 5000×5000 test case is 1555 seconds, which translates to a throughput of $1.209*10^6$ gates per second. Compared with the computing speed of 96000 gates per second [5], the acceleration rate is 12.5 fold.

The memory space saving garbling outputs for one task is 80MB (host and GPU), and circuit static information for each GC-slot costs 286MB host memory, L_O and L_I in one task is less than 20MB. The overall memory usage for computing is around 400MB. Because we observe the bursty of garbling outputs pushing into the network transferring queue, we set an empirical memory upper bound 3.2GB for the network transferring queue to prevent memory exhaustion. In sum, the total memory usage is around 3.6GB. Next, we will discuss the design for the SW problem, especially the part which is different from that for the ED problem.

```
Smith-Waterman(α, β, gap, score):
1: DP[i][0] = 0; (i=[0, α.length])
2: for j from 0 to β.length:
3:   DP[0][j] = 0; (j=[0, β.length])
4: for i from 1 to α.length:
5:   for j from 1 to β.length:
6:     signed tmp = DP[i-1][j-1] + score[α[i]][β[j]];
7:     m = 0;
8:     for o from 1 to i, and then 1 to j:
9:       m = max(m, signed (DP[x][y]-|gap(o)|) ),
         here {x,y}={i-o,j}or{i,j-o}, DP[x][y] >= |gap(o)|
10:    DP[i][j] = max(m, signed tmp);
```

Fig. 5. The SW algorithm

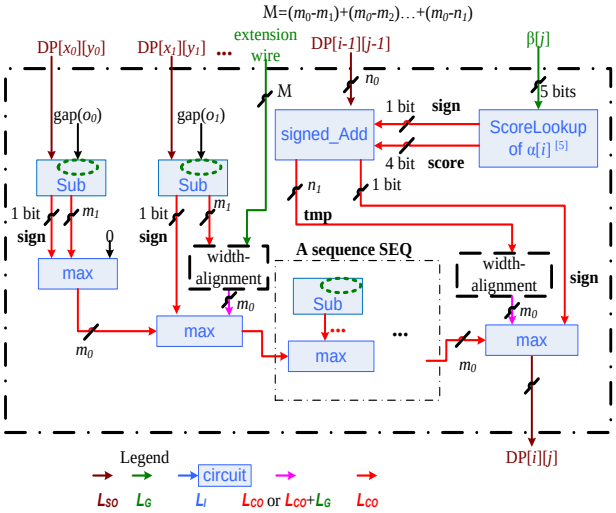

Fig. 6. The SFE building block (a GC-slot at *DP[i][j]*) for the SW problem.

IV. Privacy-preserving Smith-Waterman Computing

A. Logic of the privacy-preserving Smith-Waterman problem

The Smith-Waterman (SW) algorithm is widely used in the alignment of genome and protein sequences. In Fig. 5, we re-organize the SW algorithm into the time complexity $O(N^2)$ steps (line 6) and the time complexity $O(N^3)$ steps (lines 7-10). The algorithm inputs include two sequences α and β from the generator and the evaluator respectively, a function gap(x) = $a+b\,x$ (where a and b are public) and a 2-dimensional score matrix. Following [5], we use the typical function gap(x)=-12-7x, and BLOSUM62 [35] score matrix. There are 20 types of genome enumerated in BLOSUM62, and thus the bit width of each symbol in α and β is 5.

Lines 6-10 in Fig. 5 are translated to the building block (a

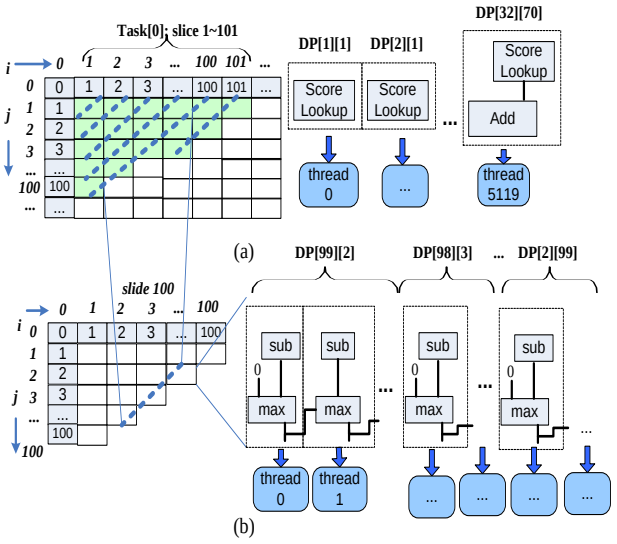

Fig. 7.Generator's GPU resource mapping policies (SW) for (a) line 6 in Fig. 5 and (b) lines 7-10 in Fig. 5

GC-slot at DP[i][j]) for the SFE of the SW problem, and the design is depicted in Fig. 6. Line 6 is implemented by a *scoreLookup* circuit of α[i] [5] and a *signed_Addition* circuit. Lines 7-10 correspond to a sequence of {*signed_Subtraction*, *Max*} circuits, denoted by *SEQ*, and an additional *Max* circuit to compare the result of SEQ and that of the *signed_Addition*. Note that the *Max* is unsigned. The original length of SEQ is $i+j$, which can be reduced based on the fact that DP[i][j] is always greater than 0. Reduction of the SEQ length is computed as follows. The maximum possible value of DP[i][j] is min(i,j)**SMAX* where *SMAX* is the maximum positive value in BLOSUM62, a pair of {*signed_Subtraction*, *Max*} circuits is discarded if min(i,j)**SMAX* <=(12+7o), for arbitrary o in [*1*,i] and then in [*1*,j]. For DP[1][j] and DP[i][1], the building block is simplified as one *scoreLookup* circuit since DP[i-1][j-1]=0, and the outputs of *Max* circuits are always 0.

For the *signed_Subtraction* circuit, because gap(o) is a real value calculated by the generator, some gates are independent to the evaluator, and thus are treated as the generator's input wires [5]. For all *Max* circuits in a GC-slot, their output bit width should be aligned as follows. Given an array of slots DP[i-o][j] (o=1,…, i) and DP[i][j-o] (o=1,…, j) which are inputs of the SEQ, we first identify the maximum possible values of DP[i-1][j] and DP[i][j-1], and then select the slot with higher maximum possible value as DP[x_0][y_0], in order to set the output width of the first *Max*, m_0 as the largest among all *Max* in SEQ. The total number of 1-bit extension wires M is equal to the sum of bit width differences of all *Max* circuits in SEQ, plus m_0 - n_1, where n_1 is the bit width of the *signed_Addition* circuit's output.

B. Resource mapping policies

For reasons similar to design of the SFE for the ED problem, CSMP and slice-by-slice policies are employed for the generator and evaluator, respectively. Fig. 7 (a) illustrates a snapshot of task[0] that contains 5120 GC-slots. The generator first garbles {*scoreLookup*, *signed_Addition*} circuits for each GC-slot in task[0]. For each slice in task [0], it calculates the number of paired {*signed_Subtraction*, *Max*} circuits of all GC-slots per slice. Then, each pair of {*signed_Subtraction*, *Max*} circuits of the current slice is mapped to one GPU gate garbler thread. Fig. 7 (b) illustrates the mapping of lines 7-10 of the 100th slice.

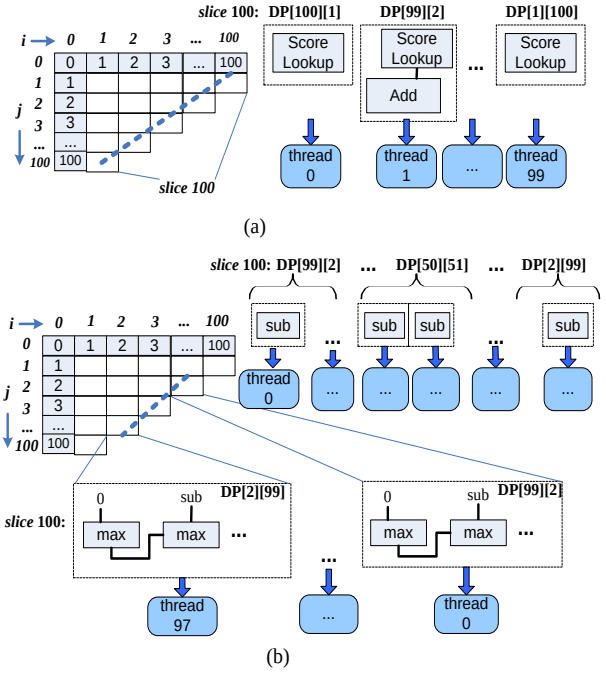

Fig. 8.Evaluator's GPU resource mapping policies (SW) for (a) line 6 in Fig. 5 and (b) line 7-10 in Fig. 5

The evaluator has a slice-by-slice resource mapping policy. Fig. 8 (a) illustrates mapping of 100 {*scoreLookup*, *signed_Addition* (if applicable)} circuits in the 100th slice to 100 GPU threads. Then, for all *signed_Subtraction* circuits in the same slice, each circuit is mapped to one GPU thread because they are independent of each other. Later, in the same slice, all *Max* circuits of one GC-slot are mapped to one GPU thread to enforce serial de-garbling of *Max* circuits.

Regarding classification of paired wire labels, L_{SO} represents the set of paired labels for wires of GC-slots' outputs, L_{CO} the set of paired labels for wires of circuits' outputs within GC-slots, L_I the set of paired labels for wires within circuits. L_I also include labels for extension wires for *Max* in SEQ and evaluator-independent gates in *signed_Subtraction*, since the number of these two types of wires in the entire DP matrix is not small. L_G includes sets of wires labels for the overflow bits of the *signed_Addition* circuit of all GC-slots in the DP matrix, and the evaluator's input β[N]. L_{CO} and L_I are pre-assigned at initialization of a new task, but L_{SO} and L_G at initialization of the SFE system. And L_{SO} is treated as global variables during the entire privacy-preserving computing because their dependency crosses the DP matrix. The static memory allocation for L_{SO}, L_{CO}, L_G and L_I is similar to its counterpart in the ED problem. The only difference is the assessment of the maximum memory usage for saving L_{CO} and L_I for line 6 is per task, and L_{CO} and L_I for line 7-10 per slice.

C. Experimental results

A 60×60 SW problem is used as the test case, which needs to be run as one task by the generator, and as 119 slices by the evaluator.

TABLE III. EXECUTION TIME BREAK DOWN (SW)

Exec Time	Generator	Evaluator
SFE system initialization	2.6s	4.51s
Housekeeping	0.0176	0.014s
GPU garbling & de-garbling(*scoreLookup*)	0.02s	0.0018s
GPU garbling & de-garbling(*signed_Addition*)	0.044s	0.037s
GPU garbling & de-garbling(*signed_Subtract*)	0.45s	0.091s
GPU garbling & de-garbling(*Max*)	2.7s	8.5s
Total computing latency	5.6s	8.64s

Table III shows that the slowest computing path is *Max*, it is mainly because all *Max* circuits within one GC-slot have to be de-garbled sequentially. The time latency from the generator's task 0 to the evaluator's de-garbling of slice 119 is 9.69 seconds, and the total computing latency (two initialization phases + 9.69, excluding networking cost) is 16.79s. This result represents a 24.7x acceleration factor over the computing time (415 seconds) for the same 60×60 SW problem reported in Huang *et al.* [5]. For the studied case, it took about 40MB to store encrypted truth table entries and

permute-and-encrypt bits. The statically allocated memory for saving all paired wire labels of GC-slots' and circuits' outputs is less than 4MB.

V. Related Work

Fairplay [1] proposed a programming language SFDL to describe the semantic of a circuit, and a low-level language SHDL to describe logic gates' inter-connection within one circuit. TASTY [2] allowed user to customize homomorphic encryption based arithmetic circuits [31] or garbled circuits to construct privacy preserving applications. Several circuit garbling techniques have been proposed: *free-XOR* [8], "*permute-and-encrypt*" [1], the *m*-to-*n* garbled lookup table and the *compact circuit* design [5], the garbled row reduction (GRR) [24] to reduce the network communication time. Jha et al. [4] proposed 3 protocols for the ED and the SW problems. Their protocol-3 solved a 200×200, 8-bit alphabet ED (60×60 SW) problem in 658 (1000) seconds. Later, Huang *et al*. [5] proposed a compact circuit design for SFE, which can compute a 2000×10000 8-bit alphabet ED problem in 223 minutes, and a 60×60 SW problem in 415 seconds. Others had optimized DP problems in the malicious model [6, 12].

Frederiksen *et al*. [34] parallelized OT protocol and multiple instances of one garbled circuit in the malicious model. CUDASW++ [9] is an open source project for plain-text large scale SW problem on GPU. We adopted the slice-by-slice mapping policy in CUDASW++ for our *slice-by-slice* mapping policy on the evaluator side, but the garbling operations are too complicated to consider adoption or modification of CUDASW+ to meet privacy preserving requirement.

VI. Conclusion

This paper presents a parallel computing model for *Secure Function Evaluation* (SFE) on massively parallel GPU architecture to solve large scale Edit Distance (ED) and Smith-Waterman (SW) problems. The experimental results showed that we achieve the highest acceleration for both ED and SW problems reported in the literature. Although our designs aimed for ED and SW problems, the two SFEs contain a set of generic arithmetic modules which can be further generalized for other types of computing tasks. A natural extension of this effort is creation of a tool chain to support automatic circuit structural information parsing, memory usage assessment and GPU resource mapping.

References

[1] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. "Fairplay – a Secure Two-party Computation System". In 13th *USENIX Security Symposium*, 2004.

[2] W. Henecka, et al. "TASTY: Tool for Automating Secure Two-Party Computations". In CCS 2010.

[3] Y. Huang, L. Malka, D. Evans, and J. Katz. "Efficient Privacy-preserving Biometric Identification". In NDSS 2011.

[4] S. Jha, L. Kruger, and V. Shmatikov. "Towards Practical Privacy for Genomic Computation". In S&P 2008.

[5] Y. Huang, *et. al* "Faster Secure Two-Party Computation Using Garbled Circuits". In 20th *USENIX Security Symposium*, 2011.

[6] B. Kreuter, A. Shelat, C.h. Shen, "Billion-Gate Secure Computation with Malicious Adversaries", In 21th *USENIX Security Symposium*,2012.

[7] Z. Erkin, M. Franz. J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. "Privacy-preserving Face Recognition". In PET 2009.

[8] V. Kolesnikov and T. Schneider. "Improved Garbled Circuit: Free XOR Gates and Applications". In ICALP 2008.

[9] Y. Liu, et al. "CUDASW+2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions". *BMC Research Notes* 3(1) 93, 2010.

[10] GTX-680 white papers. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.

[11] S. Bromlings, S. MacDonald, J. Anvik, J. Schaeffer, D. Szafron, K. Tan. "Pattern-based Parallel Programming". In ICPP 2002.

[12] Y. Huang, J. Katz, D. Evans. "Quid Pro Quo-tocols: Strengthening Semi-Honest Protocols with Dual Execution", In S&P 2012.

[13] A. Yao. "How to Generate and Exchange Secrets". In 27th *Annual Symposium on Foundations of Computer Science* (SFCS), 1986.

[14] Y. Lindell and B. Pinkas. "A Proof of Security of Yao's Protocol for Two-Party Computation". J. Cryptol., Vol. 22(2): pp. 161-188, 2009.

[15] M. Naor, B. Pinkas, and R. Sumner, "Privacy-preserving Auctions and Mechanism Design",In *ACM Conference on Electronic Commerce*,1999.

[16] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. "Efficient Privacy-preserving Face Recognition". In ICISC 2009.

[17] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. "SCiFI: a System for Secure Face Identification". In S&P 2010.

[18] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. "Extending Oblivious Transfers Efficiently". In *Advances in Cryptology* – Crytpo, 2003.

[19] M.O. Rabin. "How to Exchange Secrets with Oblivious Transfer". Technical Report 81, Harvard University, 1981.

[20] M. Naor and B. Pinkas. "Efficient Oblivious Transfer Protocols". In SODA 2001.

[21] V. Kolesnikov and R. Kumaresan. "Improved Secure Two-Party Computation via Information-Theoretic Garbled Circuits". In 8th *Conference on Security and Cryptography for Networks*, 2012.

[22] D. Harnik, Y. Ishai, E. Kushilevitz, and J.B. Nielsen. "OT-combiners via Secure Computation". In 5th *Theory of Cryptography Conference*, 2008.

[23] C. Hazay and Y. Lindell. "Efficient Secure Two-party Computation: Techniques and Constructions". Springer, 2010.

[24] B. Pinkas, T. Schneider, N. Smart and S. Williams. "Secure Two-party Computation is Practical". In *Advances in Cryptology* –Asiacrypt, 2009.

[25] C. Cachin. "Efficient Private Bidding and Auctions with Oblivious Third Party". In *CCS* 1999.

[26] Multi-precision Integer and Rational Arithmetic C/C++ Library. http://www.shamus.ie/index.php?page=Downloads

[27] Schneier on Security: "Cryptanalysis of SHA-1", http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

[28] K. Jang, S.Han, S. Han, S.Moon, K.S. Park. "SSLShader: Cheap SSL Acceleration with Commodity Processors". In NSDI 2011.

[29] Performance Comparison: Security Design Choices. http://msdn.microsoft.com/en-us/library/ms978415.aspx

[30] Crypto++5.6.0 benchmarks on Intel Pentium 4 CPU. http://www.cryptopp.com/benchmarks-p4.html

[31] C. Gentry. "a Fully Homomorphic Encryption Scheme". Ph.D. dissertation. http://cs.au.dk/~stm/local-cache/gentry-thesis.pdf.

[32] SHA-256 in PolarSSL. https://polarssl.org/sha-256-source-code

[33] K. Asanovic, *et al.* "The Landscape of Parallel Computing Research: A View from Berkeley", tech report UCB/EECS-2006-183, 2006.

[34] T.K. Frederiksen, J.B. Nielsen. "Fast and Malicious Secure Two-Party Computation Using the GPU". http://eprint.iacr.org/2013/046.pdf.

[35] S. Henikoff and J.G. Henikoff. "Amino Acid Substitution Matrices from Protein Blocks". In the *National Academy of Sciences of the United States of America*, 1992.

[36] J. Guilford, K. Yap, V. Gopal. "Fast SHA-256 Implementations on Intel Architecture Processors". http://download.intel.com/embedded/processor/whitepaper/327457.pdf