OT Communicative and Computational Benchmark Tests

Choosing among Ferret [1], Silver [2], Silent OT [3] and Softspoken OT [5] for Hardware Acceleration

Oblivious Transfer

About OT:

1. Sender does not know which message is requested by receiver
2. Receiver can only know the message requested
3. A very primitive, simple protocol with huge applications, namely MPC

Why OT?

1. Information is secure when multiple parties jointly compute something
2. Integrate into more complex protocols to improve runtime or mitigate certain constraints

1-out-of-2 Regular OT

Sender

Receiver

m_0, m_1 — b

RSA Keygen

N, e, d → N, e

Random strings

x_0, x_1 → x_0, x_1

v ← $(x_b + k^e)\ \%\ N$

$k_0 = (v-x_0)^d\ \%\ N$
$k_1 = (v-x_1)^d\ \%\ N$

$n_0 = m_0 + k_0$
$n_1 = m_1 + k_1$ → n_0, n_1

$m_b = n_b - k$

Random OT

Sender

Receiver

Correlated OT

Sender

Receiver

Computation-Communication Scale

LPN Assumption … Decoding a certain LDPC structure

Less computation

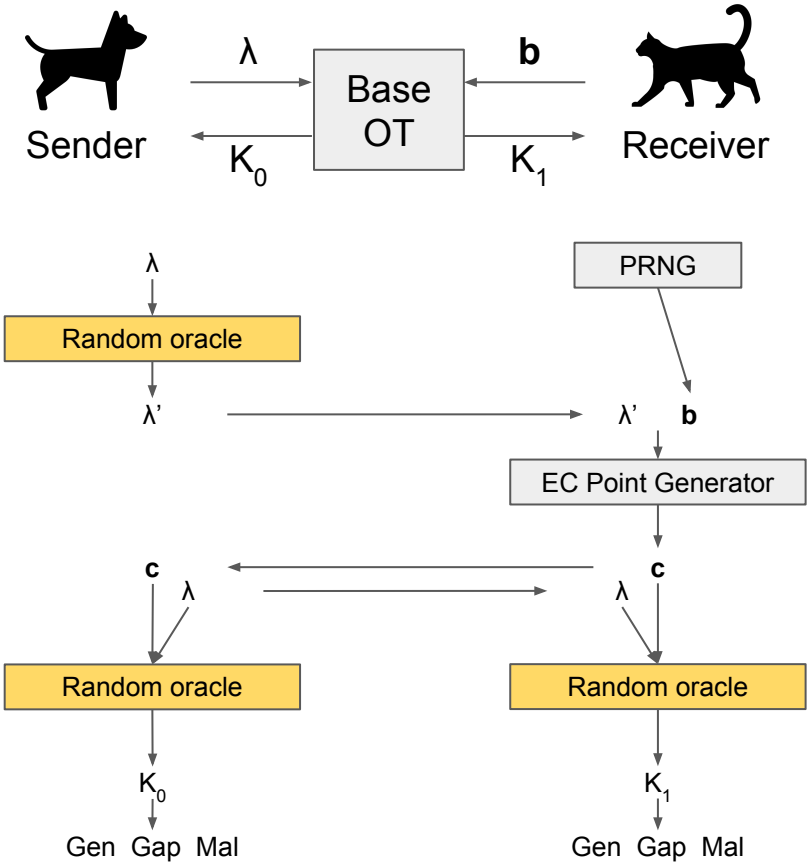
More communication

More computation

Less communication

Computation: (in ms)
Communication (in kB)

Silent OT – phase 1: seed generation



Stage	Action	Sender	Action	Receiver
Gen base OT	malloc	SilentOtExtSender::gen SilentBaseOts Allocate array to hold baseOT content	rand	SilentOtExtReceiver::genSilentBaseOts Randomly generate an array of choice bits
PCG construction	comm	SimplestOT::send Input seed into random oracle and send output Receive new randomised string Send seed to receiver	comm	SimplestOT::receive Receive sender's ro output, obfuscate selector bits into string and send it back Receive seed from sender
	mult	Update random oracle with randomised string and seed and use that as correlated random string for later	mult	Update random oracle with randomised string and seed and use that as correlated random string for later
Set base OT	memcpy	SilentOtExtSender::set SilentBaseOts Partition base OTs into gen, gap and malicious checking	memcpy	SilentOtExtReceiver::setSilentBaseOts Partition base OTs into gen, gap and malicious checking

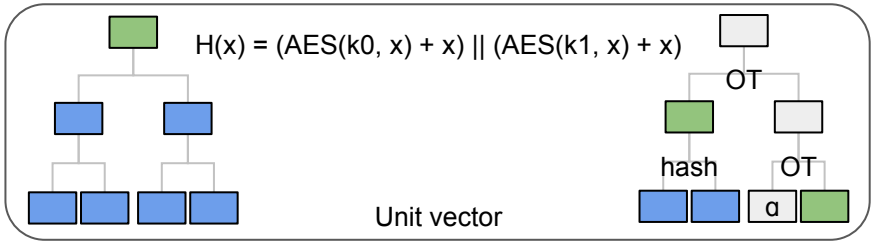
Silent OT – phase 2: silent expansion

Sender

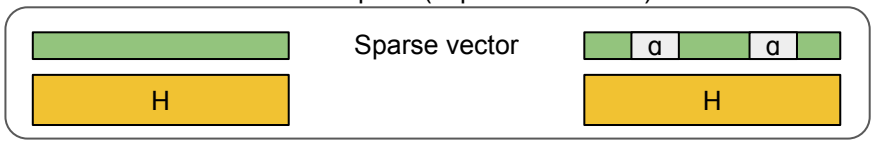
Silent Exp

Receiver

PPRF GMM tree expansion (parallel)



LPN assumption (improved in Silver)



Pseudorandom vector

Stage	Action	Sender	Action	Receiver
Gap value	comm	SilentOtExtSender::silentSendInplace Send correlated gap value to receiver	comm	SilentOtExtReceiver::silentReceiveInplace Receive correlated gap value from sender
GMM tree construction	malloc	SilentMultiPprfSender::expand Allocate t number of GMM trees with depth log(domain / key len)	encrypt	SilentOtExtReceiver::silentReceiveInplace Derandomise OTs by encrypting gap values
Tree node expansion	hash	SilentMultiPprfSender::Expander::run Use k0 to hash parent node to child nodes	malloc	SilentMultiPprfReceiver::expand Allocate t number of GMM trees with depth log(domain / key len)
	comm	SilentMultiPprfSender::Expander::run Send masked sums of each level	comm	SilentMultiPprfReceiver::Expander::run Receive base OT string with masked sums
			hash	SilentMultiPprfReceiver::Expander::run Hash parent nodes

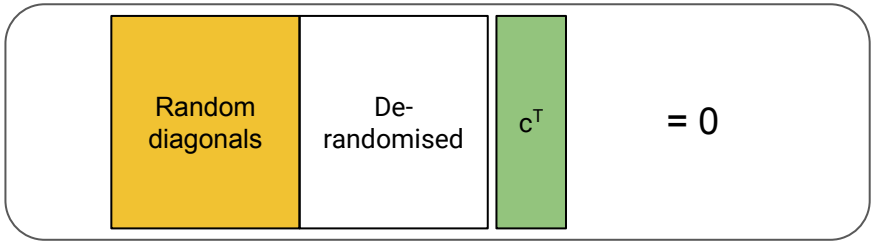
Silver code - decoding LDPC matrix

Sender

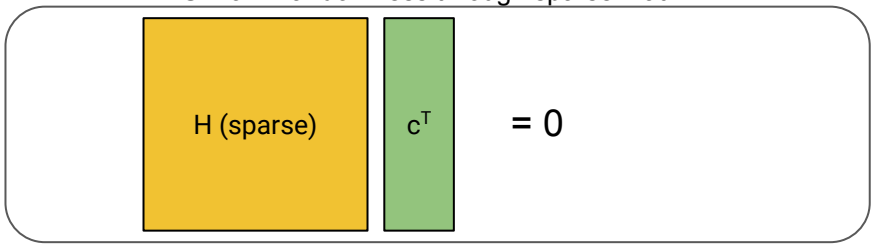
LDPC Hash

Receiver

Uniform randomness through Silver LDPC matrix



Uniform randomness through sparse matrix



Stage	Action	Sender	Action	Receiver
LDPC compress	mult	SilentOtExtSender::compress() Encode the output matrix in quasi-cyclic mode	mult	SilentOtExtReceiver::compress Encode the output matrix in quasi-cyclic mode
LDPC hash	hash	SilentOtExtSender::silentSend Hash the ldpc encoding using AES blocks to obtain messages	hash	SilentOtExtReceiver::silentReceive Hash the ldpc encoding using AES blocks to decrypt only the selected messages

Improvements of Silver LDPC matrix over sparse random matrices

- Min distance between 1s
- Quick sparse matrix multiplication due to de-randomised right
- Defeats even the comparatively efficient IKNP extension

FERRET - How It Differs From Silent OT

Similar or identical
Phase 1: Seed Generation and Sharing
Phase 3: Silent Expansion
Phase 4: Symmetric OT

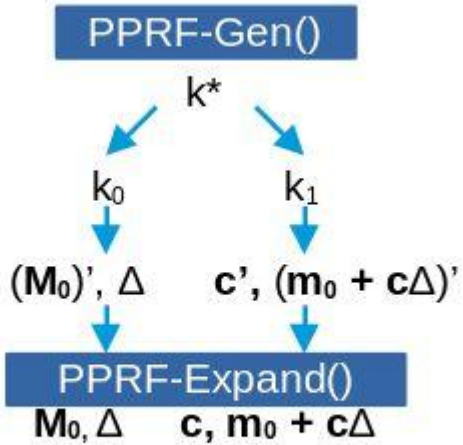
Phase 2: PCG construction with no overhead on multi-point COT
Use Cuckoo hashing
Ensure consistency of the correlation values
LPN amplifier
New check against malicious adversaries

SILVER - How It Differs From Silent OT

Similar or identical
Phase 1: Seed Generation and Sharing
Phase 4: Symmetric OT

Phase 2-1: PCG construction with linear test framework
LPN can be exploited: i.e. gaussian elim
Use a linear test framework to guarantee resilience against attacks
Phase 2-2: ALT-g encoder
Phase 2-3: Decoding LDPC
Phase 2-4: Matrix with derandomised left:main diagonal being non-zero and derandomised right for minimum linear distance and efficient memory

Phase 3: Silent Expansion
OLE – instead of selection being binary, it is a field element
Ex: f = ax + b; receiver may send {(a, b) | a, b∈F}
VOLE – OLE but the messages and field selectors are vectors

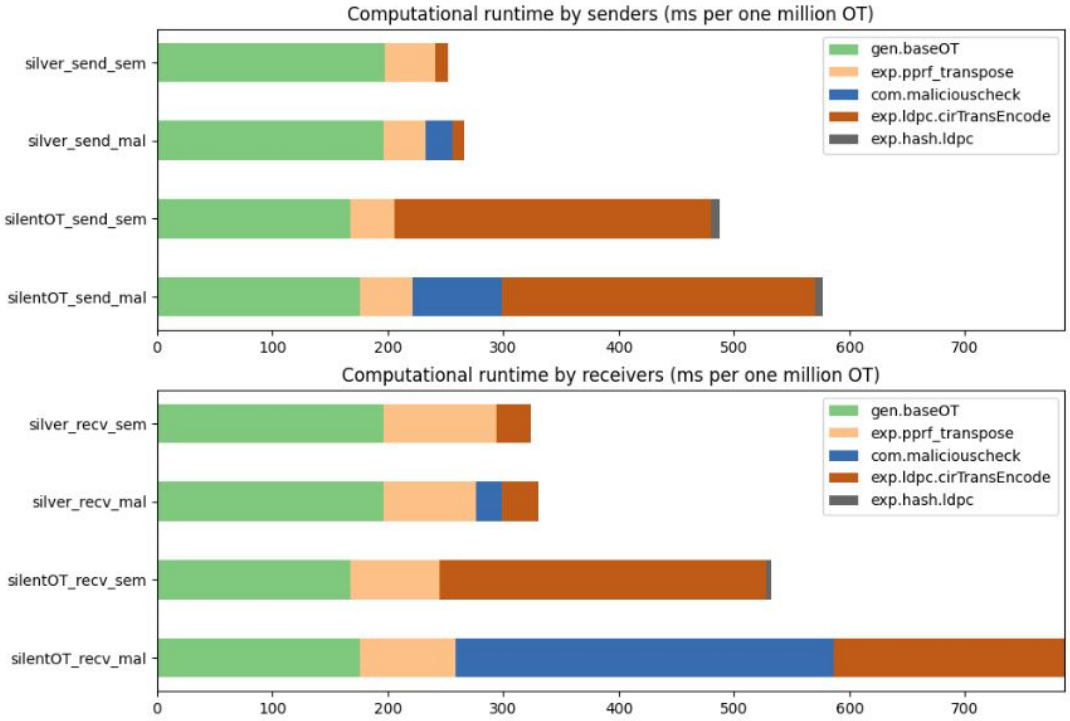


Note: c can be a field element

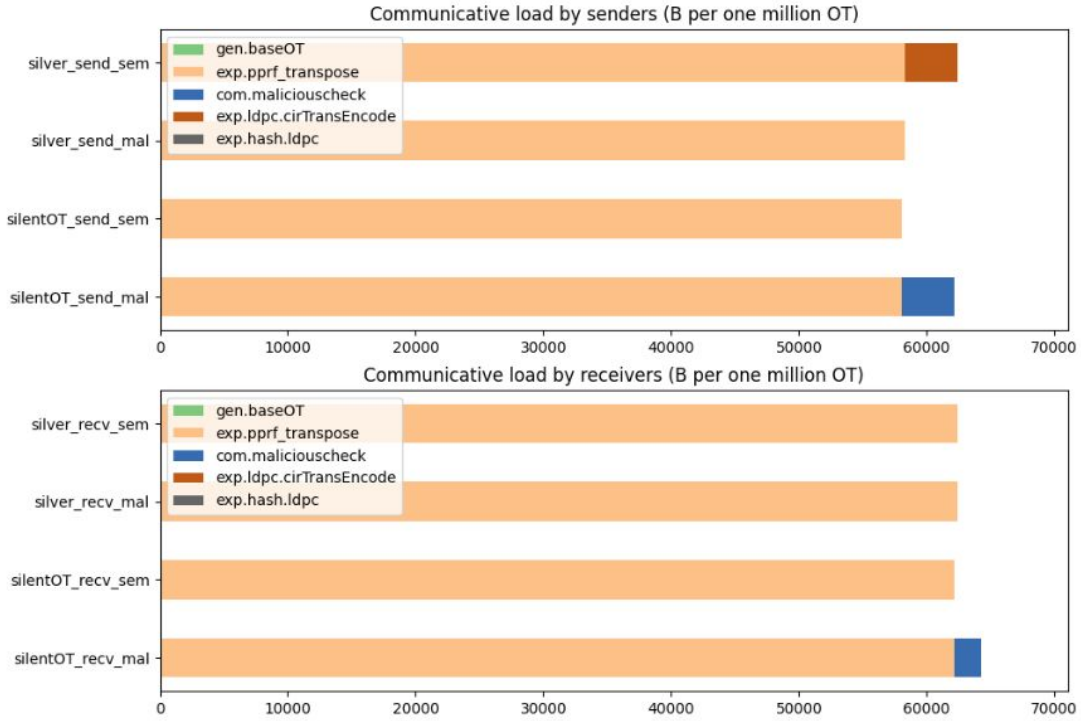
Table Summary of Enhancements

Previous works such as IKNP, Silent OT	Softspoken	Ferret	Silver
Seed generation			
Single-point COT			
Multi-point COT		Less overhead with Cuckoo hashing	
LPN assumption		New consistency check against malicious entities LPN amplifier	Enhanced security using unique encoding procedure for LDPC

Benchmark Results for Computation



Benchmark Results for Communication



Summary

There is a tradeoff between computation and communication among these OT protocols. Selecting the most optimal one to use often depends on bandwidth, network setup, and hardware specifications.

FERRET optimises certain steps of the PCG construction, such as the malicious checking and multi-point COT.

SILVER makes use of a stronger LPN assumption that is tested against the Linear Test Framework and a half sparse half diagonal configuration to encode the LDPC matrix.

Softspoken OT is a compromise between computation and communication, and has a variable setting ‘k’ to allow for tuning between the two extremes.

What have we looked at / can be improved?

- Alternative hashing algorithm during tree expansion
- Parallelisation of tree expansion on GPU
- Sparse matrix multiplication during LDPC encoding

Block Ciphers for Constrained Hardware

Three best alternatives to AES

Simon – bitwise operations
Speck – modular operations
Chaskey – ARX (addition, round key, xor) operations

According to the benchmark conducted by Dinu et. al, Chaskey is the all-round better block cipher in terms of code size, ram usage and time taken during cipher. [10]

Meanwhile, Simon and Speck can have blocksize and keysize tweaked to perform better on hardware constrained devices such as microcontrollers.[9]

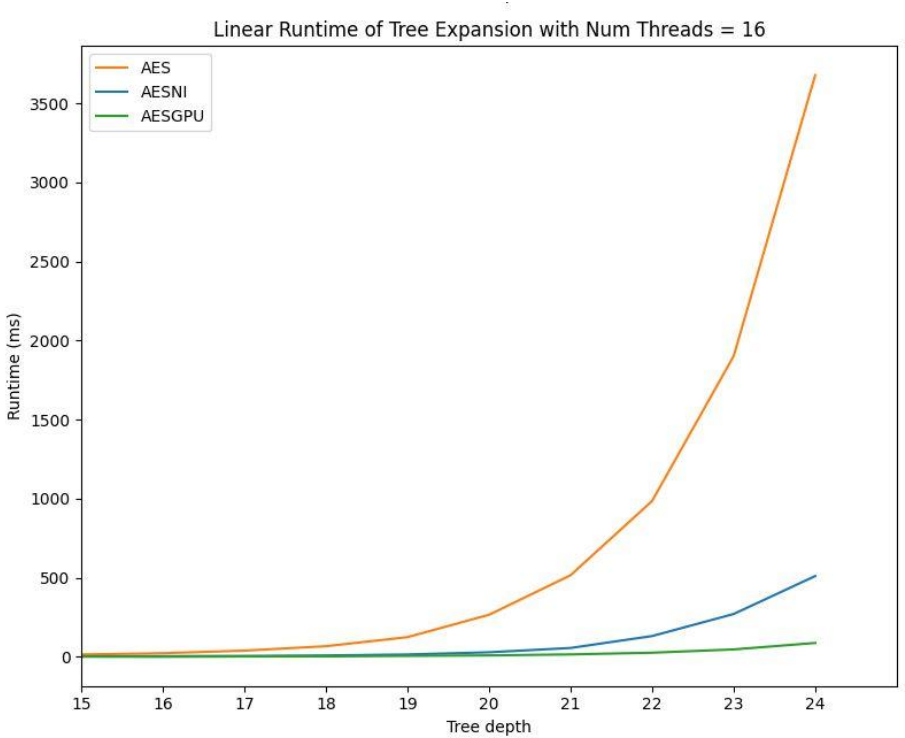
Speedup on GPU for AES Hash Tree Expansion

Tree expansion runtime

what worked
1. reduce host to device memcpy by keeping tree on device
2. early initialisation of cuda system via cudaFree
3. having each thread write to tree node directly
4. early one-time allocation of all buffers using precomputed size
5. using pinned memory during host array allocation

what didnt work
1. async memcpy to host after a layer is expanded
why not: kernel execution too quick, memcpyasync would still bottleneck runtime
2. cudaMallocManaged
why not: slowed down runtime due to on-demand copying

https://github.com/Andrew-Gan/aes-pprf-expander



Tree Expansion is Compute Bound

CUDA API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
31.5	686,163,267	384	1,786,883.5	3,032	30,187,607	cudaDeviceSynchronize
30.4	664,139,754	418	1,588,851.1	4,373	42,066,583	cudaMemcpy
27.9	607,824,544	1	607,824,544.0	607,824,544	607,824,544	cudaHostAlloc
5.2	113,635,372	1	113,635,372.0	113,635,372	113,635,372	cudaFreeHost
2.5	53,617,627	34	1,576,989.0	12,047	4,534,739	cudaMalloc
2.4	53,033,494	34	1,559,808.6	7,565	10,537,297	cudaFree
0.1	2,901,084	768	3,777.5	2,907	39,786	cudaLaunchKernel
0.0	44,660	2	22,330.0	3,214	41,446	cudaCreateTextureObject
0.0	11,768	2	5,884.0	3,019	8,749	cudaDestroyTextureObject

CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	662,237,563	768	862,288.5	6,240	14,703,030	aesExpand128(unsigned long long, TreeNode*, unsigned int*, unsigned long, unsigned long)

CUDA Memory Operation Statistics (by time):

Time(%)	Total Time (ns)	Operations	Average	Minimum	Maximum	Operation
96.4	661,308,207	16	41,331,762.9	41,210,999	42,043,385	[CUDA memcpy DtoH]
3.6	24,418,514	384	63,589.9	2,656	786,083	[CUDA memcpy DtoD]
0.0	41,728	18	2,318.2	1,984	2,944	[CUDA memcpy HtoD]

Sparse Matrix Multiplication on GPU

Silver LDPC matrix uses a diagonal representation with derandomised right side.

According to Bell et. al, matrix can be represented in data-offset format to fetch nearby data array values in one iteration. [7]

A paper titled ‘spECK’ on using lightweight analysis to determine best sparse matrix packed representation before performing matrix multiplication [11]

Row Analysis	Symbolic SpGEMM	Local Load Balancing	Global Load Balancing	Numeric SpGEMM	Sort Output Data
I. Total number of products II. Number of products in longest row III. Min and max col idx of referenced rows	I. Count number of non-zero results to determine output matrix size II. Compute row offsets in CSR	I. Thread assignment to rows to balance out workload II. More threads per row allow mem coalescing	I. Binning rows to be collectively handled by a thread block II. Choose hash size that maximises scratchpad memory	I. Make use of earlier analysed parameters, compute and store products II. Should be more efficient than naive implementation	I. Optional, done if the non-zeros produced by step V is unsorted

References

[1] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang, ‘Ferret: Fast Extension for Correlated OT with Small Communication’, in Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event USA, Oct. 2020, pp. 1607–1626. doi: 10.1145/3372297.3417276.

[2] G. Couteau, P. Rindal, and S. Raghuraman, ‘Silver: Silent VOLE and Oblivious Transfer from Hardness of Decoding Structured LDPC Codes’, in Advances in Cryptology – CRYPTO 2021, vol. 12827, T. Malkin and C. Peikert, Eds. Cham: Springer International Publishing, 2021, pp. 502–534. doi: 10.1007/978-3-030-84252-9_17.

[3] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl, ‘Efficient Pseudorandom Correlation Generators: Silent OT Extension and More’, in Advances in Cryptology – CRYPTO 2019, vol. 11694, A. Boldyreva and D. Micciancio, Eds. Cham: Springer International Publishing, 2019, pp. 489–518. doi: 10.1007/978-3-030-26954-8_16.

[4] “iptables(8) - Linux man page,” Iptables(8) - linux man page. [Online]. Available: https://linux.die.net/man/8/iptables. [Accessed: 30-Oct-2022].

[5] L. Roy, ‘SoftSpokenOT: Communication–Computation Tradeoffs in OT Extension’, p. 42.

[6] L. Roy, “Communication-Efficient Secure Two-Party Computation From Minimal Assumptions.” Oregon State University.

[7] N. Bell and M. Garland, ‘Efficient Sparse Matrix-Vector Multiplication on CUDA’, p. 32.

[8] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, ‘Sparse matrix solvers on the GPU’, p. 8.

[9] R. Beaulieu, D. Shors, and J. Smith, ‘Simon and Speck: Block Ciphers for the Internet of Things’. National Security Agency, Jul. 09, 2015. Accessed: Nov. 25, 2022. [Online]. Available: https://eprint.iacr.org/2015/585.pdf

[10] D. Dinu, Y. L. Corre, D. Khovratovich, L. Perrin, J. Großschädl, and A. Biryukov, ‘Triathlon of lightweight block ciphers for the Internet of things’, J Cryptogr Eng, vol. 9, no. 3, pp. 283–302, Sep. 2019, doi: 10.1007/s13389-018-0193-x.

References, cont.

[11] M. Parger, M. Winter, D. Mlakar, and M. Steinberger, ‘spECK: accelerating GPU sparse matrix-matrix multiplication through lightweight analysis’, in Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego California, Feb. 2020, pp. 362–375. doi: 10.1145/3332466.3374521.

[12] Gpgpu-Sim, “ISPASS2009-benchmarks/AES at master · GPGPU-SIM/ISPASS2009-benchmarks,” GitHub. [Online]. Available: https://github.com/gpgpu-sim/ispass2009-benchmarks/tree/master/AES. [Accessed: 02-Mar-2023].