Peningkatan Kinerja Modul Pencocokan Pola dalam Sistem Deteksi Intrusi Snort Menggunakan GPU

Afrizal Fikri
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
afrizalf96@gmail.com

Achmad Imam Kistijantoro
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
imam@informatika.org

***Ringkasan*—Kemajuan teknologi telah memberikan berpengaruh pada pertumbuhan pesat penggunaan data digital. Di era digital ini, banyak transaksi penting juga dilakukan melalui dunia maya. Mayoritas dari itu adalah data konfidensial yang rahasia. Dengan demikian, jaminan keamanan menjadi sangat penting. Salah satu titik masuk dari penyalahgunaan data berasal dari sisi server. Penjaminan keamanan pada server yang menggunakan sistem deteksi intrusi jaringan (NIDS) dapat menghabiskan sumber daya dan waktu yang banyak. Salah satu bagian penting dari NIDS adalah pencocokan string dalam tahap analisis. Berdasarkan masalah ini, makalah ini akan membahas eksperimen untuk meningkatkan kecepatan pencocokan string untuk memaksimalkan kinerja keseluruhan sistem menggunakan GPU. Berdasarkan percobaan dan pengujian, solusi yang diusulkan memiliki waktu operasi yang jauh lebih baik dibandingkan dengan solusi CPU saat ini dengan tetap menjaga akurasi.**

***Index Terms*—pattern matching; intrusion detection; GPU; parallel computation; CUDA**

I. Pendahuluan

Salah satu sumber daya yang penting untuk dilindungi yaitu *server*. Banyak aspek security yang terdapat pada *server* dan dapat dengan mudah diserang [1]. Solusi untuk pengamanan *server* dari *request* berbahaya adalah dengan menggunakan sistem deteksi intrusi jaringan atau *network intrusion detection system* (NIDS). NIDS berfungsi mengenali kemungkinan serangan dari *request* yang diterima pada *server* atau *client* sebagai tindakan pencegahan. Ketika *request* dianggap berbahaya, maka dapat dilakukan tindakan lanjutan untuk mencegah kerusakan lebih lanjut pada aset. NIDS yang melakukan tindakan preventif seperti ini disebut juga NIPS (*network intrusion prevention system*).

Karena adanya peningkatan kecepatan *traffic* internet dan banyaknya serangan yang terjadi, maka dibutuhkan NIDS yang mampu melakukan deteksi dengan lebih cepat. Salah satu *bottleneck* dalam pengecekan paket adalah banyaknya *rule* yang harus dicocokkan [2]. Sehingga, peningkatan secara signifikan dapat dicapai salah satunya dengan meningkatkan kemampuan komputasi untuk pencocokan banyak paket secara paralel.

Komputasi paralel dapat dilakukan dengan *multicore* CPU [3], *multicore* GPU [4], atau prosesor terkustomisasi seperti ASIC dan FPGA [5]. Perbedaan antara CPU dan GPU adalah bagaimana mereka memproses pekerjaan [6]. CPU didesain untuk melakukan komputasi secara serial dengan beberapa *core*, sementara GPU dapat memiliki ribuan unit komputasi yang disebut *stream processor* (SP) yang bekerja secara paralel.

Berdasarkan tinjauan di atas, penelitian ini akan membahas metode yang dapat digunakan dalam mempercepat pencocokan string pada NIDS dengan memanfaatkan *multithreading* pada GPU. Pengujian akan dilakukan untuk mengukur peningkatan kinerja terhadap solusi CPU yang digunakan oleh NIDS saat ini.

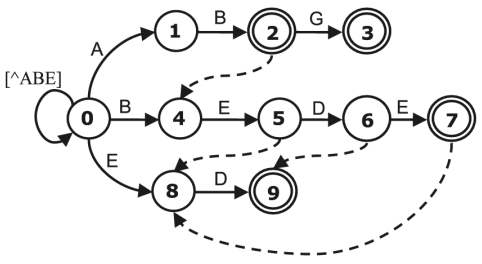
II. Pencocokan string

Secara umum, ada dua jenis pencocokan string: *single pattern string matching* dan *multi pattern string matching*. Kami akan fokus ke pencocokan *multi pattern string matching*. Ada beberapa algoritma terkenal untuk *multi pattern string matching*, seperti algoritma Aho-Corasick [7], Commentz-Walter [8], dan Wu-Manber [9]. Implementasi akan berbasis variasi dari algoritma Aho-Corasick.

A. Algoritma Aho-Corasick

Algoritma Aho-Corasick adalah jenis algoritma pencocokan string yang bekerja dengan mencocokkan tiap string ke kamus. Kamus akan berisi semua pola untuk dicocokkan dalam bentuk *state machine*.

Dalam algoritma ini, operasi pencocokan dilakukan dengan menelusuri *state machine* untuk tiap karakter dalam string input. Jika sampai pada *final state*, maka string input dianggap cocok dengan salah satu pola. Pencocokan berlanjut hingga string input berakhir.



Gambar 1. Example state machine of Aho-Corasick dictionary.

Aho-Corasick menggunakan *failure function* untuk melakukan *backtrack* setelah berhenti di salah satu *final state*. *Failure function* akan menunjuk dari *final state* sekarang ke prefiks pola terpanjang yang cocok dengan sufiks dari pola yang saat ini dicocokkan [7]. *Failure function* mengurangi redundansi untuk sekuens yang telah dilewati. Dengan metode ini, semua pola yang ada dalam kalimat dapat dicocokkan dengan sekali penelusuran dan mengurangi kompleksitas dari $O(mn)$ to $O(n)$.

Dalam algoritma ini, akses memori menjadi *bottleneck*. Setiap operasi penelusuran akan mengambil satu entri dari tabel. Karena akses memori lebih mahal daripada komputasi, algoritma ini terbatas dengan memori (*memory bound*) [10].

B. Data Parallel Aho-Corasick

Untuk membuat Aho-Corasick menjadi *multithreading*, beban pencocokan perlu dibagi ke setiap *thread*. Salah satu caranya adalah dengan mempartisi input menjadi beberapa segmen dan kemudian setiap *thread* akan mencocokkan tiap segmen yang berkorespondensi terhadap kamus. Namun, jika ada pola yang menjangkau beberapa segmen sekaligus, pola tidak akan dikenali pada kedua *thread*. Masalah ini dikenal sebagai *boundary matching problem*. Sehingga perlu memperpanjang sesuai dengan panjang pola terpanjang. Akses memori meningkat menjadi $O((n/s + m) * s) = O(n + ms)$ dengan s adalah banyak segmen.

Gambar 2. Pattern **AB** not recognized by both thread 3 and 4

C. Parallel Failureless Aho-Corasick

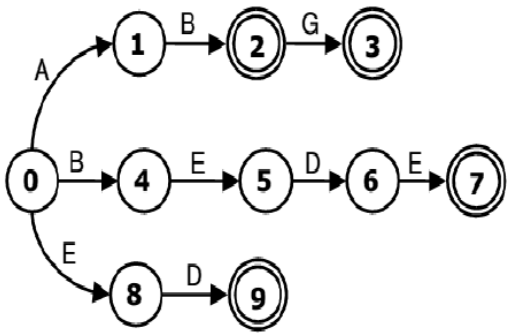
Metode lain untuk mengadaptasi Aho-Corasick dengan *multithreading* adalah membagi *thread* ke semua *byte* karakter. Setiap *thread* akan mencocokkan pola yang dimulai dengan karakter yang berkorespondensi dan selesai ketika ada *final state* atau tidak ada transisi yang valid pada karakter [10]. Konsekuensinya adalah setiap *thread* paling banyak hanya akan cocok dengan satu pola.

Dengan demikian, *failure function* tidak diperlukan lagi. Selain itu, *boundary matching problem* tidak akan terjadi dengan menggunakan pendekatan ini. Pendekatan ini dinamakan juga sebagai *parallel failureless Aho-Corasick* (PFAC).

III. Related Works

A. Utilization of Double Buffering Scheme, Texture Memory and Pinned Memory

Dalam tulisan ini, implementasi pencocokan berbasis GPU telah diusulkan menggunakan algoritma Aho-Corasick. Selain pemanfaatan GPU multithreading, ada beberapa penyesuaian untuk mengimpor Aho-Corasick ke GPGPU. Mesin negara dibangun menggunakan tabel transisi pada tabel 2D



Gambar 3. State machine without failure function

pada memori tekstur. Berdasarkan eksperimen, penerapan ini meningkatkan kinerja sebesar 19% [4].

Untuk skema transfer memori, skema *double buffering* diusulkan. Paket akan di-batch dalam satu *buffer*. Setiap kali *buffer* penuh, semua paket ditransfer ke GPU dalam satu operasi. Mentransfer dari *perangkat* ke *host* juga dilakukan dengan cara yang sama. Semua hasil akan dikumpulkan di *buffer* kedua dan ditransfer ketika penuh.

Trik optimasi lain yang digunakan adalah menggunakan *pinned memory*. Kedua *buffer* in *host* akan manggunanakan *pinned memory* untuk mengurangi swappiness. Dan juga *pinned memory* memungkinkan transfer secara langsung tanpa melalui CPU atau *direct memory access* (DMA) dan mengurangi kebutuhan komputasi dan latensi. Desain ini dapat meningkatkan kinerja sistem hingga 3,2 kali lebih tinggi daripada implementasi menggunakan *multithreading* pada CPU.

B. Scalable Architecture for Maximizing GPU Utilization

Dengan profiling, disimpulkan bahwa *bottleneck* terjadi pada 3 komponen: akuisisi paket, pencocokan string, dan pencocokan aturan opsi oleh *regular expression*. Selain itu, khususnya di dalam komponen pencocokan string, rata-rata utilisasi GPU masih rendah. Penelitian ini mencoba menerapkan perbaikan arsitektur pada NIDS. Diantara metode yang diusulkan adalah *pipelining* dan *payload batching*. *Pipelining* digunakan untuk memisahkan penggunaan *thread* untuk transfer dan pencocokan. Dan dengan arsitektur ini, NIDS dapat ditingkatkan ke beberapa GPU dan CPU *core* inti [11].

Perbaikan yang dicapai dengan metode ini adalah sekitar 1,5 hingga 4 kali lebih baik dibandingkan dengan solusi *baseline* Snort. Evaluasi dilakukan menggunakan CPU ganda Intel X5680 dengan masing-masing 12 core dan GPU ganda NVIDIA GTX 580. NIDS dijalankan di bawah *traffic* jaringan dengan kapasitas sekitar 40 Gbps dan bisa mencapai *throughput* 25,2 Gbps.

C. Failureless Aho-Corasick by Maximizing Shared Memory Usage

Salah satu cara untuk melakukan pencocokan Aho-Corasick adalah dengan menetapkan setiap *byte* input ke setiap *thread* secara bersamaan. Kemudian setiap *thread* akan melakukan

pencocokan yang dimulai dengan *byte* yang ditetapkan. Pencocokan berhenti setiap kali masuk *final state* atau transisi valid tidak ada. Pendekatan ini tidak memerlukan *failure function*. Sehingga setiap *thread* hanya bertanggung jawab untuk mencocokkan paling banyak satu pola [10].

Pendekatan baru ini memanfaatkan *memory coaelescing* dalam GPU sehingga *byte* yang berurutan dapat dimuat sekaligus. Untuk lebih meningkatkan kinerja, transisi dari *state* awal akan dimuat ke *shared memory* untuk mengurangi transaksi ke *global memory*.

Karena tabel transisi dapat sangat besar, *shared memory* tidak dapat menampung tabel transisi. Sehingga, sebagai alternatif untuk tabel transisi akan diikat ke *texture memory*. *Texture memory* memiliki beberapa kelebihan seperti memiliki *cache* lebih besar dan cocok untuk melakukan akses dengan pola akses yang sulit diprediksi. Pengujian menggunakan dataset PCAP dari DEFCON menunjukkan peningkatan kinerja yang signifikan. Menggunakan CPU Intel Core i7-950 dan GPU NVIDIA GTX 580, *throughput* sistem bisa menjadi 3 kali dibandingkan dengan Snort *baseline*.

IV. Proposed Solution

A. Matching Algorithm

Untuk mengimplementasikan Aho-Corasick ke GPU, kita perlu mengeksploitasi SIMD (*single instruction multiple data*) secara optimal. Pendekatan data paralel adalah salah satu pendekatan yang paling mudah untuk diterapkan. Masukan ada dipartisi menjadi beberapa segmen. Setelah memisahkan segmen, masing-masing segmen akan dijalankan di *thread* berbeda dengan ekstensi sepanjang pola terpanjang. Pendekatan ini memiliki 2 masalah: ekstensi meningkatkan transaksi data secara drastis, dan tidak optimal jika masukan dipisah menjadi banyak segmen. GPU dapat membangkitkan ratusan *thread* dan dengan demikian semua *thread* dapat dimanfaatkan secara optimal.

Alternatif lain adalah dengan menggunakan metode yang disebut PFAC. Ide mengalokasikan tiap *thread* ke setiap *byte* dari aliran input memiliki implikasi penting pada efisiensi *state machine* PFAC.

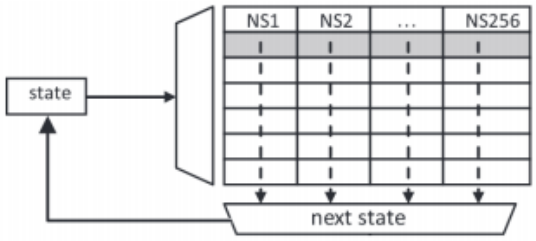
1) Tiap *thread* hanya bertanggung jawab dengan string yang dimulai dengan karakter pada *thread* tersebut. Ketika tidak ada pola yang cocok dengan huruf pada *thread*, pencarian langsung berhenti pada *thread*. Akibatnya, umur *thread* juga lebih singkat.
2) Dalam mesin PFAC, tiap 32 *thread* dalam *warp* akan mengakses memori yang berurutan dari memori global. Dengan demikian, akses tabel transisi fitur *memory coalescing* dapat dimanfaatkan.
3) Tidak memerlukan penyimpanan tambahan untuk *failure transition*. Sehingga konsumsi memori akan lebih sedikit dan kemungkinan *cache hit* lebih besar.

B. *Implementasi* State Machine

Ada 2 alternatif untuk implementasi mesin negara: tabel dua dimensi atau trie dengan pointer. Struktur trie mendukung modifikasi dinamis. Tetapi implementasinya terlalu rumit untuk disimpan di GPU. Dan juga memori bisa jadi terfragmentasi dan tidak dapat mengeksploitasi *spatial locality*. Selain itu biaya mengalokasikan memori di GPU sangat besar. Sehingga opsi ini kurang disukai.

Desain lain dan yang lebih mudah adalah tabel dua dimensi. Untuk masing-masing *state*, akan ada transisi ke, katakanlah 256 karakter (dengan tipe input adalah karakter ASCII *unsigned*). Model ini sederhana dan mendukung *memory coalescing*. Jika kita menggunakan *texture memory*, juga akan ada peluang pola akses yang akan di-*cache*.

Setelah semua pola terdaftar, tabel akan dikompilasi di *host*. Dan kemudian, tabel ditransfer ke *perangkat* sebanyak ukuran tabel yang terisi. Gambar. 4 di bawah ini menunjukkan desain tabel transisi.



Gambar 4. Transition table representation

Untuk mengurangi kebutuhan terhadap penanda status akhir, nomor status akan disusun ulang. Status akhir akan diletakkan pada status pertama hingga ke-N. Sedangkan status lainnya termasuk status awal dimulai dari N+1. Dengan demikian, kebutuhan memori untuk penanda akan berkurang. Selain itu, operasi pengecekan dalam kode *kernel* lebih sederhana dan ringan.

C. *Alokasi* Thread

Meskipun *thread* dibangkitkan sangat banyak, banyak juga diantara *thread* yang akan berhenti sangat awal. Sehingga, muatan pencocokan tiap *thread* dapat timpang cukup jauh dan utilisasi GPU rata-rata menjadi rendah. Untuk menunjang utilisasi GPU, [10] mengusulkan alokasi memori yang berulang.

Dalam satu blok, satu atau beberapa *byte stream* akan ditempatkan sebagai lokasi awal satu *thread*. Misal, dalam blok berisi 4.096 *byte* dengan 512 *thread*, *thread* pertama akan memulai pencocokan pada posisi kelipatan 512. Akibat dari penugasan yang berulang, secara statistik kemungkinan ketimpangan muatan dari masing-masing *thread* akan berkurang [10].

D. *Optimasi pada Memori GPU*

Untuk memaksimalkan utilisasi GPU dan mengurangi latensi akibat transfer data, beberapa metode dapat diterapkan. Salah satu metode yang sangat umum adalah penggunaan *shared memory*. *Shared memory* memiliki latensi yang lebih kecil beberapa kali lipat dibandingkan *global memory*. Namun, ukuran *shared memory* sangat terbatas. Selain itu, *shared memory* harus di-"replikasi" ke setiap *block*.

Dengan memanfaatkan sifat *early terminate* pada PFAC, penggunaan *shared memory* dapat dihemat. Transisi dari *state* awal akan selalu diakses namun kemungkinan pencocokan berhenti sangat tinggi [10]. Sehingga transisi ini akan dimuat ke *shared memory* untuk menurunkan latensi ketika pembacaan tabel transisi.

Sisa tabel transisi yang masih ada di *global memory* akan ditingkatkan dengan *texture memory*. Diantara keuntungan *texture memory* adalah memiliki *cache* yang lebih besar dan mampu melakukan transpolasi sehingga dapat memanfaatkan *cache* untuk pola yang susah ditebak [10]. Kelemahan *texture memory* data ditulis hanya dari *host*, cocok dengan skema pencocokan yang melakukan kompilasi kamus sekali dalam satu *rule group*.

Metode lainnya yaitu dengan melakukan *batching* dan menyimpan batch dalam *pinned memory*. *Pinned memory* adalah memori yang dialokasi di *host* dan tidak dapat di-*swap* ke disk. Penggunaan *pinned memory* untuk *buffer* input dapat menurunkan kemungkinan swappiness untuk ukuran *buffer* yang besar. Selain itu, *pinned memory* juga memungkinkan adanya DMA dari *host* ke *device* maupun sebaliknya [4].

Penggunaan *thread* per *block* dibatasi sebesar 32 *thread* untuk mencegah *bank conflict*. Ukuran *bank* pada *shared memory* pada GPU NVIDIA 940MX yang memiliki arsitektur Maxwell yaitu 32. Jika akses ke *shared memory* lebih besar dari ukuran *bank*, operasi akan diserialisasi dan berdampak ke latensi secara signifikan [10].

V. Implementasi pada Snort

Dalam mengimplementasi modul pencocokan string, akan digunakan *interface* MPSE (*multi pattern search engine*). Modul deteksi pada Snort akan memuat semua kelas yang mengimplementasi *interface* MPSE. Pilihan metode pencocokan akan diberikan dari konfigurasi sesuai definisi pada file header.

Interface MPSE sendiri berisi beberapa fungsi yang digunakan untuk analisis sebagaimana pada Tabel I berikut.

Tabel I
Fungsi yang diimplementasi

Nama fungsi	Deskripsi
init	Menginisiasi struktur data dan melakukan alokasi
register	Mendaftarkan pola ke dalam larik, sekaligus memperbarui jumlah *state* maksimal
compile	Membentuk kamus berdasarkan pola yang telah didaftarkan
match	Melakukan pencocokan string ke kamus yang dibentuk
cleanup	Melakukan dealokasi dan menghapus semua informasi pencocokan

Kemudian juga diimplementasi struktur data *handle* yang akan menampung tabel dan larik yang diperlukan oleh MPSE. Isi dari *handle* milik PFAC yaitu:

1) Larik untuk daftar pola yaitu *array of string*
2) Tabel transisi
3) Tabel temporer untuk menampung daftar pola
4) Larik untuk hasil pencocokan berupa *array of integer*
5) Jumlah *state* maksimal
6) Jumlah *state* efektif
7) Jumlah *final state*

VI. Pengujian

A. Strategi Pengujian

Pengujian dilakukan untuk mendapatkan perbandingan kinerja antara modul yang dikembangkan dan modul yang ada dalam NIDS Snort. Pengujian dibatasi dengan menggunakan konfigurasi *data acquisition* PCAP dengan masukan *file* tcpdump saja. Berkas didapatkan dari arsip DEFCON tahun 2017. *Live testing* tidak diujikan dalam Tugas Akhir ini mengingat kapasitas *network interface* dapat menjadi *bottleneck* sehingga hasil perbandingan tidak akurat.

Pengujian dilakukan pada komputer dengan spesifikasi seperti pada Tabel II berikut.

Tabel II
Spesifikasi Lingkungan Pengujian

Komponen	Spesifikasi
Sistem Operasi	Ubuntu 16.04.5 amd64
CPU	Intel Core i7-7500U CPU @ 2.70GHz × 4
Host RAM	16 GB
Media Penyimpanan	SSD SATA III 6 Gbps
GPU	NVIDIA GeForce 940MX
Arsitektur GPU	Maxwell
Compute Capability	5.0
Dedicated Video RAM	2 GB
CUDA *Runtime*	CUDA 8.0 r2
C/C++ *Compiler*	GCC 5.4.0

Ada 5 skenario yang akan diujikan dalam penelitian ini:

1) *Baseline* (Aho-Corasick dengan *multithreading* CPU)
   Baseline akan menggunakan konfigurasi default Snort. Ini menjadi dasar pengukuran kinerja dan kebenaran program yang diimplementasikan pada GPU.
2) Skenario 1 (PFAC dengan *global memory*)
   Skenario ini adalah skenario paling dasar. Optimasi hanya dilakukan pada algoritma tanpa melibatkan optimasi pada latensi GPU.
3) Skenario 2 (PFAC dengan *shared memory*)
   Skenario ini memanfaatkan *shared memory* untuk mengurangi akses ke *global memory*. Transisi *state* awal ke tiap huruf serta *stream* masukan ditampung dalam *shared memory*.
4) Skenario 3 (PFAC dengan *shared memory* dan *pinned memory*)
   Skenario ini mirip dengan skenario 2 dengan tambahan *pinned memory* pada *buffer*. Mekanisme ini diharapkan dapat mengurangi *swappiness* dan memungkinkan DMA (*direct memory access*).
5) Skenario 4 (PFAC dengan *shared memory*, *pinned memory*, dan *texture memory*)
   Skenario ini mirip dengan skenario 3 dengan tambahan *texture memory* pada kamus. Dengan *cache* yang lebih besar, penggunaan *texture memory* diharapkan mengurangi transfer memori.

B. Hasil Pengujian Kinerja

Pengujian dilakukan dengan menjalankan Snort dengan konfigurasi *data acquisition* PCAP, yaitu membaca berkas PCAP dan menggunakan *single thread* untuk implementasi GPU dan 4 *thread* untuk implementasi bawaan Snort. *Ruleset* yang digunakan yaitu Snort VRT 3000 yang didapat dari laman Snort. Pengujian dilakukan untuk ukuran *buffer* yang bervariasi, yaitu 128 kB, 512 kB, 1024 kB, dan 2018 kB.

Tabel III
HASIL PENGUJIAN

Skenario	*Buffer* (kB)	Runtime (detik)	Speedup
Baseline	128	94.632	1
	512	92.656	
	1024	90.118	
	2048	96.097	
Skenario 1	128	450.629	0.21
	512	463.281	0.2
	1024	500.656	0.18
	2048	436.805	0.22
Skenario 2	128	29.207	3.4
	512	39.321	3.16
	1024	31.29	2.88
	2048	31.302	3.07
Skenario 3	128	31.44	3.01
	512	38.446	2.41
	1024	35.202	2.56
	2048	31.404	3.06
Skenario 4	128	30.825	3.07
	512	29.137	3.18
	1024	22.417	4.02
	2048	20.446	4.7

Hasil skenario 1 memilki kapasitas paling rendah. Alasannya yaitu karena operasi pencocokan *string* adalah operasi yang *memory-bound*. Diperlukan banyak akses ke penyimpanan kamus untuk mendapatkan transisi ke *next state*. Karena kamus disimpan di *global memory*, maka *fetch* ke *global memory* menjadi sering dan menimbulkan latensi yang cukup besar.

Skenario 2 secara umum lebih cepat beberapa kali daripada skenario 1, yaitu sekitar 15 kali lipat lebih cepat. Hal ini karena kecepatan akses *global memory* lebih besar dari *shared memory*. Dalam skenario ini, *input string* seharusnya tidak berpengaruh banyak karena adanya *coalescing access*. Sehingga pemuatan baris transisi dari *state* awal ke *shared memory* memiliki pengaruh yang besar terhadap *throughput* sistem. Ini menunjukkan bahwa mayoritas *thread* akan berhenti pada awal pencocokan.

Skenario 3 tidak terlalu berpengaruh terhadap skenario 2. Keuntungan *pinned memory* tidak terlalu terlihat bahkan cenderung memperlambat. Sejalan dengan eksperimen yang dilakukan [4]. Ini karena *pinned memory* memiliki *overhead* saat alokasi dan dealokasi. Kemungkinan besar ini terjadi karena ukuran *buffer* belum terlalu besar. *Overhead* saat alokasi masih belum sebanding dengan perbedaan *transfer* antara DMA dan melalui CPU. Selain itu, juga tidak terjadi *swap* terhadap *page* yang terdapat *buffer* karena ukuran memori masih kecil.

Sedangkan skenario 4 baru terlihat perbedaan signifikan terhadap skenario 2 dan 3 pada *buffer* sebesar 1 MB ke atas. Terlihat bahwa adanya *cache* pada memori tekstur membantu menurunkan akses ke memori global dan meningkatkan kinerja keseluruhan modul.

VII. KESIMPULAN

Peningkatan kinerja pencocokan string pada IDS Snort menggunakan GPU berhasil dilakukan dengan cara mengubah pendekatan algoritma yang digunakan agar memaksimalkan utilisasi GPU. Implementasi dilakukan berbasis teknik PFAC. Teknik ini memanfaatkan kemampuan GPU untuk membangkitakan banyak *thread* serta *memory coalescing* ketika mengakses karakter yang bersebelahan dalam *stream*. Operasi pencocokan string dari kamus merupakan operasi yang *memory bound* sehingga dilakukan beberapa teknik untuk mengurangi *latency gap* antara akses I/O, memori dan komputasi GPU. Di antara teknik yang digunakan yaitu penggunaan *pinned memory* pada *buffer input* dan *texture memory* pada kamus. Selain itu, digunakan juga penggunaan *shared memory* untuk menampung *buffer input* dan tabel transisi dari *state* awal untuk menghemat *fetch* pada memori global. Parametrisasi dapat dilakukan pada ukuran *buffer input*. Dengan menambah ukuran *buffer input*, jumlah pengiriman paket dari *host* ke *device* dapat berkurang dan dapat menurunkan latensi secara drastis.

PUSTAKA

[1] OWASP, "OWASP Top 10 Application Security Risks - 2017," 2017. [Online]. Available: https://www.owasp.org/index.php/Top_10-2017_Top_10

[2] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for Accelerating SNORT IDS," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ser. ANCS '07. New York, NY, USA: ACM, 2007, pp. 127–136. [Online]. Available: http://doi.acm.org/10.1145/1323548.1323571

[3] B. Haagdorens, T. Vermeiren, and M. Goossens, "Improving the Performance of Signature-based Network Intrusion Detection Sensors by Multithreading," in *Proceedings of the 5th International Conference on Information Security Applications*, ser. WISA'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 188–203. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31815-6_16

[4] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," in *Recent Advances in Intrusion Detection*. Springer, Berlin, Heidelberg, Sep. 2008, pp. 116–134.

[5] A. Das, D. Nguyen, J. Zambreno, G. Memik, and A. Choudhary, "An FPGA-Based Network Intrusion Detection Architecture," *IEEE Transactions on Information Forensics and Security*, vol. 3, no. 1, pp. 118–132, Mar. 2008.

[6] NVIDIA, "CUDA Toolkit Documentation," p. 19, Jan. 2017, *NVIDIA Developer Zone - CUDA C Programming Guide v8.0*. Section 3.1.5. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[7] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975. [Online]. Available: http://doi.acm.org/10.1145/360825.360855

[8] B. Commentz-Walter, "A string matching algorithm fast on the average, tr 79.09.007," *Information Systems*, vol. 5, no. 3, p. 245–246, 1980.

[9] S. Wu and U. Manber, "Fast Text Searching: Allowing Errors," *Commun. ACM*, vol. 35, no. 10, pp. 83–91, Oct. 1992.

[10] C.-H. Lin, C.-H. Liu, L.-S. Chien, and S.-C. Chang, "Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs," *IEEE Trans. Comput.*, vol. 62, no. 10, pp. 1906–1916, Oct. 2013.

[11] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: A Highly-scalable Software-based Intrusion Detection System," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 317–328.