Titre: Title:	Filtrage de contenus numériques connus à haute vitesse optimisé sur plateforme GPU
Auteur: Author:	Jonas Lerebours
Date:	2012
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Lerebours, J. (2012). Filtrage de contenus numériques connus à haute vitesse optimisé sur plateforme GPU [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1001/

UNIVERSITÉ DE MONTRÉAL

FILTRAGE DE CONTENUS NUMÉRIQUES CONNUS À HAUTE VITESSE OPTIMISÉ SUR PLATEFORME GPU

JONAS LEREBOURS
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
DÉCEMBRE 2012

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

FILTRAGE DE CONTENUS NUMÉRIQUES CONNUS À HAUTE VITESSE OPTIMISÉ SUR PLATEFORME GPU

présenté par: LEREBOURS Jonas
en vue de l'obtention du diplôme de: Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury d'examen constitué de:

M. BRAULT Jean-Jules, Ph.D., président
M. DAVID Jean Pierre, Ph.D., membre et directeur de recherche
M. SAVARIA Yvon, Ph.D., membre

« *Rendez vous compte !*
sur Internet, n'importe qui peut dire n'importe quoi ! »

Un député français à l'assemblée nationale
durant les débats sur HADOPI.

REMERCIEMENTS

J'aimerais remercier sincèrement le professeur Jean Pierre David, mon directeur de recherche, pour son suivi et ses conseils. Il a toujours su répondre à mes interrogations en pointant de nouveaux aspects, en proposant une solution innovante, m'incitant à chercher plus loin, avec toujours perspicacité et rigueur. Sa disponibilité m'a donc permis de m'améliorer sans cesse, et la confiance qu'il m'a accordée tout au long de ma maîtrise s'est avérée un précieux atout, duquel j'espère avoir pu profiter à la juste mesure.

Je souhaite également remercier le CRSNG, ainsi que NetClean et le Ministère des Finances et de l'Économie du Québec pour les financements qui ont été attribués au projet et desquels j'ai pu bénéficier. Ces financements ont permis de mener à bien les expérimentations, en travaillant sur du matériel de pointe, pour obtenir des résultats originaux et de premier plan.

Mes remerciements s'adressent aussi aux autres étudiants sous la supervision du professeur David et avec qui j'ai eu la chance de travailler : Tarek Ould Bachir, Marc-André Daigneault, Adrien Larbanet, Mathieu Allard et Patrick Grogan. Même si nos projets étaient pour certains complètement différents, ils ont toujours réussi à m'aider à prendre du recul et à débloquer des situations qui me paraissaient sans issue. Toute cette équipe a fait de notre laboratoire un espace de travail convivial. J'ai eu grand plaisir à discuter avec eux, à les connaître, et j'espère ne pas les perdre de vue.

Je remercie encore Élise Mellon, Maeva Jaffar et Vincent Lecomte, qui ont toujours été mes plus assidus supporters, pour leur soutien continu. Ils m'ont donné beaucoup de leur temps et ont considérablement apporté à ce mémoire par leurs commentaires inénarrables et plus qu'exhaustifs, fruits de relectures scrupuleuses.

RÉSUMÉ

Beaucoup de données transitent sur les réseaux informatiques tels que le réseau Internet, et une partie de celles-ci peut être illégale. Les autorités qui contrôlent un réseau donné ont souvent besoin d'empêcher la transmission de certains documents, qu'ils soient dangereux, illicites, ou simplement refusés. Une entreprise pourrait par exemple vouloir empêcher que certains documents sortent de leur réseau interne, de même que des écoles pourraient chercher à filtrer l'accès à des sites sensibles. Nous allons présenter dans ce mémoire un système de détection et de blocage qui vise à repérer le passage de contenus spécifiés sur une connexion réseau. Le but est qu'un utilisateur ne puisse jamais charger ou envoyer une image, un film ou tout autre document référencé comme interdit, quelle qu'en soit la source.

Suivant le contexte, il peut être utile de travailler à grande échelle, c'est-à-dire de contrôler le trafic d'une large population avec un système unique et centralisé. On peut alors se placer au niveau des fournisseurs d'accès et de services Internet, ou encore sur la connexion à Internet d'une très grande entreprise. On contrôle ainsi l'ensemble des communications qui entrent et sortent de leurs réseaux sans avoir à modifier la configuration des postes utilisateurs. Le nombre de ce type d'interconnexion est relativement réduit, ce qui simplifie le déploiement. En revanche, les débits en ces points sont très élevés car ils concentrent le trafic de tous les utilisateurs. C'est là qu'apparait le principal défi, car les fibres optiques utilisées aujourd'hui permettent de faire passer 40 à 100 Gb/s, ce qu'un processeur classique d'usage général n'est pas capable de traiter.

Travailler avec de tels débits de données demande ainsi une puissance de de calcul importante, et impose de réduire et optimiser les traitements au maximum. Les approches existantes pouvant filtrer autant d'information se basent souvent simplement sur les descripteurs des communications tels que l'adresse de l'émetteur ou du destinataire. Lorsqu'un site internet est suspecté d'envoyer aux clients du contenu interdit, c'est l'ensemble du site internet qui est bloqué, ou l'ensemble des connexions du client qui sont coupées, ce qui manque de précision. Dans notre cas, on cherche à bloquer la transmission de certains contenus uniquement, préalablement référencés, en laissant passer le reste du trafic. Il faut donc être capable de repérer ces contenus (images, vidéos, programmes, etc.) au milieu de l'ensemble des données échangées. De plus, dans la plupart des réseaux utilisés aujourd'hui, les données ne sont pas transmises telles qu'elles en une seule fois, mais sont d'abord divisées en plusieurs petits fragments appelés paquets, qui sont transmis indépendamment. Il faut ainsi détecter des extraits des contenus interdits parmi d'autres données fragmentées, ce qui demande une plus grande précision de détection, ceci avec plusieurs millions de paquets de

données par seconde. On bloquera ensuite uniquement la connexion qui a transmis cette partie de document.

D'autre part, nous cherchons à référencer un nombre important de documents. Le système ne doit donc pas souffrir de ralentissement lié au nombre de contenus à bloquer. Les programmes antivirus ou de détection d'intrusion, qui fonctionnent sur le même modèle d'analyse des données transmises en temps réel, se réfèrent en général à des bases de données de modèles d'attaques qui doivent être compilées avant d'être utilisées. Plus le nombre de règles à tester est important, plus la compilation est difficile et longue. Les systèmes qui se basent sur cette approche sont alors limités à quelques dizaines de milliers de règles. Notre système peut en revanche stocker un nombre théoriquement infini de documents sources, la seule limite étant la mémoire utilisée, sans impact sur les performances, grâce à un format de base de données de contenus interdits très simple et efficace.

Pour relever le défi de traiter de très hauts débits sans limiter le nombre de documents référencés, nous avons utilisé l'algorithme de *max-hashing*. Cet algorithme a été spécialement conçu pour détecter très rapidement des fragments de documents connus, en un minimum d'opérations. On distingue deux phases : le référencement des documents à détecter et bloquer, puis l'analyse en temps-réel du flux réseau. Seule cette seconde partie est étudiée ici, le référencement étant réalisé hors ligne par les autorités qui décident quels documents interdire. Nous l'avons implémenté sur processeur graphique (GPU) afin de disposer de leur immense puissance de calcul parallèle, très adaptée pour l'analyse des innombrables paquets indépendants à traiter. Nous présentons dans ce mémoire le fonctionnement de ce type de matériel et les adaptations nécessaires pour qu'un algorithme utilise au maximum les ressources disponibles et soit capable d'analyser le plus grand débit de données possible.

Nous détaillons l'implémentation de l'algorithme de *max-hashing* sur les GPU de NVidia, ainsi que les performances que nous avons pu mesurer. Un seul processeur graphique peut ainsi traiter plus de 70 Gb/s de données sources tout en référençant plus d'un million de documents à détecter. La vitesse de traitement est néanmoins limitée par le bus qui transmet les données vers la carte graphique. Après étude des modes de ce bus, on parvient à envoyer à 45 Gb/s, ce qui offre tout de même les performances suffisantes pour analyser une connexion 40GbE, et est à notre connaissance sans équivalent dans la littérature. De plus, lorsque la configuration du bus le permet, des configurations à plusieurs cartes graphiques peuvent être mises en place, multipliant les débits traités et le nombre de références enregistrées.

Des tests sur un réseau réel à 10 Gb/s ont été réalisés, en transmettant les résultats du module de détection à un logiciel dédié au filtrage. Nous avons ainsi pu mettre en place très rapidement un système complet fonctionnant en temps-réel avec une seule carte graphique. On mesure une latence minimale de 6 ms entre l'arrivée d'un paquet de données et la mise en

place du blocage si nécessaire. Ce système peut s'intégrer de façon totalement transparente sur le réseau à contrôler, et stopper la transmission de tout document interdit.

ABSTRACT

More and more data are being transmitted every day through computer networks such as the Internet. Part of these data may be illegal and networks authorities often need active filtering so that unwanted contents would not enter the network, or private documents would not leave the intranet. This thesis proposes such a filtering appliance, able to efficiently detect and block known documents passing through a watched connection. The main goal is that users could not load or send any document that would be known as forbidden, regardless of its origin.

Working at large scales can be necessary when one wants to control large populations with no need to setup filters on every connected device. One unique and centralized service deals with the whole traffic. Internet services providers are good examples as they agglomerate all their subscribers' communications. They link with each other using quite few interconnection points, which simplifies a global deployment. The first challenge is that these links use fast media with 40 to 100 Gbps bandwidths. Such high data rate cannot be handled by a standard general-purpose processor and filtering that information is therefore very challenging.

Previous approaches that are able to filter such amounts of data suffer two different limitations. First, some only use communication descriptors such as emitter and receiver addresses. When a website is suspected to send forbidden information to a network user, either the whole website or the whole user connections are blacklisted as no further investigation is possible. We ought to block specific and previously listed contents only, allowing all harmless data. We therefore need to spot these contents (images, videos, software...) among other flowing data. Moreover, network protocols require splitting sent data into small chunks, namely packets, which are transmitted independently. Thus, we more precisely need to spot extracts of the referenced contents among other fragmented data, at rates of millions of packets per second. Greater precision is required, but we can then block the only flow that is sending the spotted content.

The second limitation is the number of referenced documents. We want to detect an important set of different contents and the system cannot loose performances when this set enlarges. Antiviral or anti-intrusion systems, which are based on the same real-time analysis model, often use regular expression patterns as rules. These patterns need to be compiled before they can be used, and large rule sets make compilation slower and harder. Such systems are therefore often limited to several thousand rules. Our system can reference millions of documents with no impact on its computation speed, thanks to our simple and efficient forbidden files database format.

We describe a parallel implementation of the max-hashing algorithm that enables the detection of known content by processing network packets individually. The final system rises to the challenge of processing ultra-high bandwidths while referencing millions of documents. The target architecture is based on Graphics Processing Units (GPUs), which are known to offer tremendous performances for highly parallel applications, at low cost. The algorithm first collects a set of fingerprints from the listed documents to detect. Fingerprints are small subsets of the reference documents supposedly unique to the documents and easily identifiable. At detection time, those fingerprints are detected in the network packets and reported to an application that correlates all the matches. Results demonstrate that a single GPU board can theoretically monitor the Internet traffic up to 70 Gbps, with the ability to host hundreds of millions of reference fingerprints. Strangely, the most challenging task is to feed the board at such bandwidths through a standard interface such as PCIe. In fact, the bus cannot transfer more than 45 Gbps, which thus is the system limitation. Nevertheless multi-GPU configurations can be set up depending on the PCIe bus architecture, which multiplies the processing rate and optimize the different resources use.

A complete filtering system demonstrates the functionality of the proposed approach over a 10GbE connection. We measure a minimum latency of 6 ms and the system can be installed on network connections transparently.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

LISTE DES SIGLES ET ABRÉVIATIONS

CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
CTPH	*Context-Triggered Piecewise Hashing*
DFA	*Deterministic Finite Automaton*
DMA	*Direct Memory Access*
DPI	*Deep Packet Inspection*
FAI	Fournisseurs d'Accès à Internet
FPGA	*Field-Programmable Gate Array*
GPGPU	*General-Purpose computation on Graphics Processing Units*
GPU	*Graphics Processing Unit*
IDS	*Intrusion Detection System*
IP	*Internet Protocol*
IPS	*Intrusion Prevention System*
MAC	*Media Access Control*
MD5	*Message Digest #5*
NFA	*Non-Deterministic Finite Automaton*
PCI	*Peripheral Component Interconnect*
PCIe	PCI Express
SHA1	*Secure Hash Algorithm*
SIMD	*Single Instruction Multiple Data*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*

CHAPITRE 1

INTRODUCTION

En 2012, 2.5 exaoctets [1] de données sont créées chaque jour [32] et sont amenées à transiter sur des réseaux, notamment sur Internet. En conséquence, on mesure que chaque année le trafic Internet augmente d'environ 35% [11]. Ces chiffres peuvent paraître impressionnants, ou finalement assez modestes rapportés à la population mondiale, mais il est certain que la démocratisation du numérique entraîne un accroissement rapide des quantités de données produites et échangées. Une partie du contenu peut être illégale, la pornographie infantile en est un exemple, présenter des risques comme les virus informatiques, ou être simplement gênante, comme le spam.

Contrôler les données qui transitent sur les réseaux devient primordial pour les administrateurs. Plus encore, il est intéressant de pouvoir en bloquer la transmission pour limiter leur propagation. Les fournisseurs de service d'email ont par exemple intérêt à développer leurs contrôles pour réduire la quantité de spam car ces courriers indésirables, outre amoindrir la satisfaction des utilisateurs, consomment une très large part de leur bande passante, puisqu'ils représentent entre 65 et 75% des messages [13]. Il y a donc des intérêts tant économiques que judiciaires à contrôler le trafic sur les réseaux.

Nous allons présenter dans ce mémoire une solution de blocage de contenus connus. Nous cherchons en effet un système capable d'être inséré de façon transparente dans un réseau, c'est-à-dire sans en modifier la configuration. Il s'agira, après référencement du contenu de certains documents, images, vidéos ou autres, de repérer ces contenus parmi les informations transitant sur le segment surveillé et éventuellement de les bloquer. Le but est d'obtenir un système pouvant référencer un très grand nombre de documents différents, de l'ordre de plusieurs millions, et capable de traiter les données avec des performances, latence et bande passante par exemple, assez importantes pour pouvoir offrir ce nouveau type de contrôle à de grands réseaux d'entreprises ou à des Fournisseurs d'Accès à Internet (FAI). Travailler à ce niveau permet en effet une surveillance à très grande échelle, sans avoir à installer d'autres protections sur les postes utilisateurs, nombreux et hétérogènes. Une entreprise peut choisir de bloquer la diffusion de documents confidentiels vers l'extérieur, tandis qu'un FAI peut vouloir ou se voir imposer de bloquer la propagation de contenus illicites sur Internet.

1. exa = 10^{18}

1.1 Concepts de base

Avant de nous pencher sur la solution à mettre en place, il est important de comprendre le fonctionnement des transmissions sur Internet et les enjeux d'un tel filtrage. Nous allons donc dans un premier temps passer rapidement en revue les défis actuels dans l'analyse des communications réseaux, avant d'insister sur l'intérêt qui existe dans le filtrage de contenu. Finalement, nous étudierons le fonctionnement des réseaux et les protocoles utilisés pour l'échange des données.

1.1.1 Transmission de données

Comme nous l'avons évoqué dans le paragraphe précédent, la quantité de données qui transite en permanence sur l'ensemble des liaisons mondiales est immense. Des réseaux scientifiques ou industriels sont très utilisés pour le partage et la décentralisation des informations. De plus, la démocratisation du *cloud computing* accélère cette tendance : les fichiers et les programmes ne sont plus sauvegardés et exécutés sur l'ordinateur de l'utilisateur mais sur des serveurs distants. Ces services sont d'une grande qualité mais nécessitent une connexion à Internet puissante et imposent le transfert de beaucoup de données.

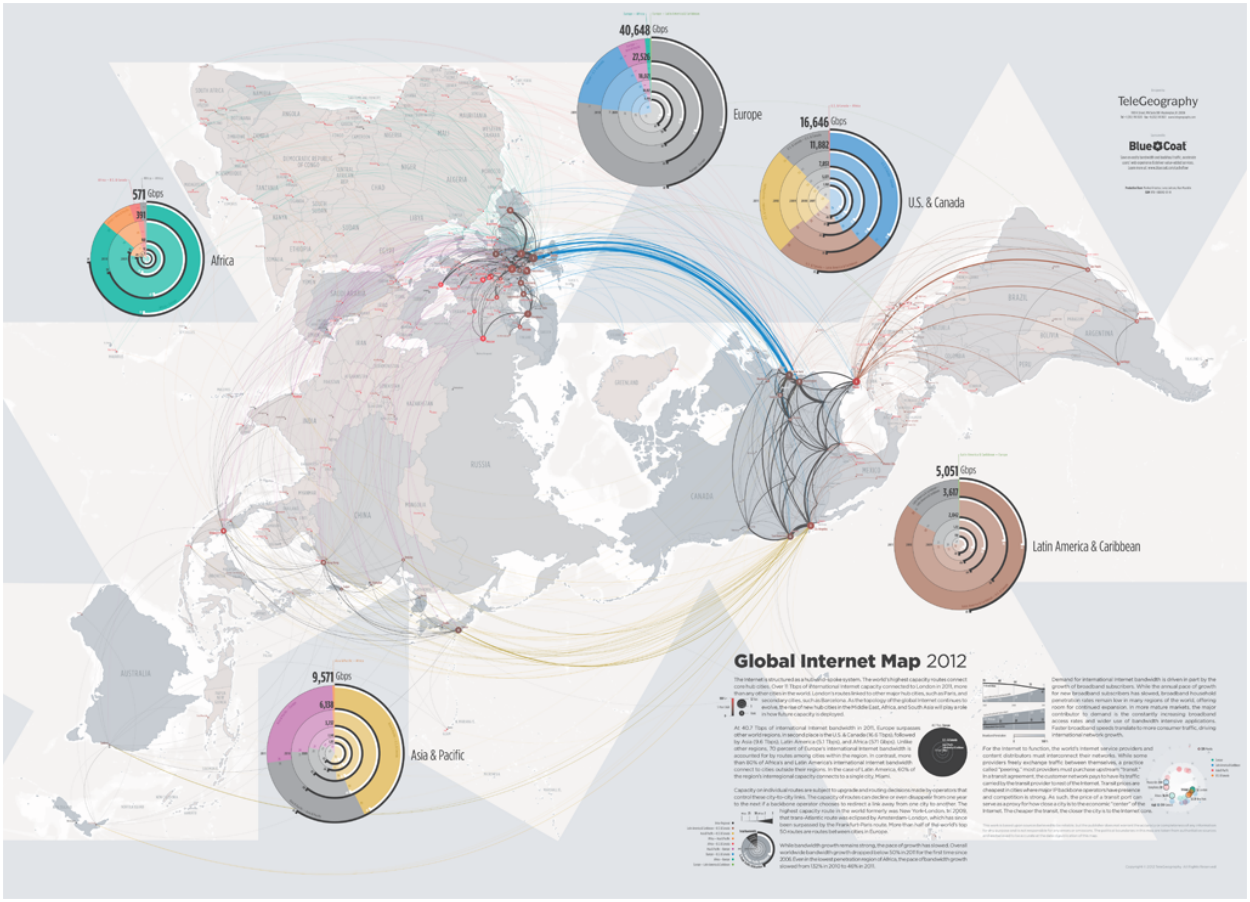
D'autre part, avec l'explosion de la connectivité (mobile comme fixe) de la population, chacun est à même de partager tout contenu, en temps réel. Les technologies évoluent parallèlement : la résolution des appareils photos augmente, produisant des images de plus en plus volumineuses, les terminaux deviennent plus puissants, ce qui permet aux fournisseurs de proposer des plateformes en ligne de plus en plus élaborées. Les bandes passantes des réseaux de communication augmentent elles-aussi, ce qui permet aux utilisateurs d'accéder aux ressources disponibles en ligne de façon similaire à celles localisées directement sur leur ordinateur. Tous ces éléments mènent à une utilisation de plus en plus grande des réseaux. De même, les entreprises, qui utilisaient historiquement des réseaux informatiques fermés pour plus de sécurité, s'ouvrent et s'étendent sur des réseaux virtuels multisites. Beaucoup de données sont donc créées et transmises, et doivent alors emprunter des voies de communications adaptées, le réseau Internet en étant la principale composante.

Plusieurs études ont regroupé des données sur les débits des communications à l'échelle mondiale. On trouve notamment sur le site Internet de TeleGeography des cartes très claires sur la connectivité internationale, telles que celle reprise sur la figure 1.1. On constate qu'en 2011 l'Europe disposait d'une capacité globale de 40 Tb/s et l'Amérique du nord de 16 Tb/s. Même en période de pointe, ces bandes passantes sont largement prévues pour le trafic actuel (47% d'utilisation en Europe, 53% en Amérique du Nord en 2011) mais ces données donnent une idée claire du défi pour contrôler l'ensemble de ce trafic.

1.1.2 Filtrage de données

Pourquoi alors vouloir contrôler tant de données ? Deux buts principaux justifient souvent cette demande. La protection des réseaux et des utilisateurs contre les contenus illégaux ou dangereux est prédominante. Par exemple, les entreprises, écoles ou autres collectivités utilisent souvent un *proxy* pour filtrer les requêtes à Internet avec une liste de sites interdits. Cela permet aussi de limiter l'accès à des sites sensibles ou la transmission de fichiers exécutables et notamment de virus. Dans un autre contexte, repérer la diffusion de contenus interdits tels que les abus sexuels sur les enfants permet aux services de police de tracer les auteurs ainsi que d'en limiter les "utilisateurs".

La copie et le plagiat menacent certains intérêts financiers en allant notamment contre le droit d'auteur. Le second but de l'inspection des flux de données est ainsi d'encadrer la reproduction incontrôlée de tout contenu. Par exemple, un éditeur peut vérifier la source des fichiers envoyés et publiés, à la recherche d'extraits de contenus existants. On peut alors



Carte par Telegeography ("*Global Internet Map 2012*")
Contenu sous licence Creative Commons BY-NC-ND 3.0

Figure 1.1 – Carte 2012 de la connectivité Internet mondiale

combattre le plagiat efficacement, voire limiter les tentations des auteurs directement en les avertissant. On met alors en place des outils d'analyse de réseau, à plus ou moins grande échelle, qui se chargent d'analyser les données transmises.

Les solutions existantes capables de travailler à de tels débits sont généralement basées sur uniquement l'analyse de la source et/ou de la destination des messages. On bloque ainsi une image ou un site Internet complet via leur adresse. Ces méthodes sont simples car elles ne s'intéressent qu'aux descriptions des transmissions, sans avoir à en reconnaître le contenu. Des systèmes de listes noires existent dans beaucoup de pays et sont imposés aux fournisseurs d'accès à Internet. Ils sont rapides à mettre en place mais ne peuvent pas combattre l'apparition de sites miroirs par exemple, ou le déménagement d'un site interdit à une autre adresse. La Chine est l'exemple le plus connu, où des lois régissent directement la liste des sites autorisés ou non pour les internautes de ce pays. Elle a cependant aussi mis en place d'autres techniques plus avancées qui permettent de terminer une connexion lorsqu'un certain nombre de mots-clés y ont été détectés. Un tel contrôle demande alors d'être capable d'analyser les communications de l'ensemble de la population.

C'est cette dernière approche qui nous intéresse, en élargissant les mots clés à tout contenu numérique. En terminant la connexion assez rapidement dès qu'un fragment interdit est repéré, on empêche tout passage d'un document complet.

1.2 Analyse des besoins

Différents besoins clés doivent impérativement être intégrés à un système de filtrage réseau. Nous allons dans cette partie les mettre en exergue afin de les étudier précisément. Nous pourrons alors les garder à l'esprit tout au long de la conception et de la validation. Une fois ces bases posées, nous pourrons justifier l'emploi des technologies choisies. Nous avons déjà évoqué les motivations et les protocoles à utiliser, nous détaillerons dans cette partie les besoins qu'ils entraînent : la détection de fragments de documents, le référencement de nombreux documents, et la détection en temps réel.

1.2.1 Détection de fragments de documents

Le réseau Internet, de même que la grande majorité des réseaux actuellement, est basé sur le protocole Ethernet. Celui-ci impose que les documents transmis soient d'abord divisés en plusieurs sous-blocs envoyés successivement. Chaque bloc peut avoir une taille quelconque inférieure à 1500 octets, et est intégré dans une "trame", qui contient en particulier l'adresse de la machine et du destinataire ainsi que la taille du bloc. Nous présenterons les protocoles utilisés dans les réseaux à la section 2.1. Notons que les trames ne sont pas nécessairement

de la taille maximale autorisée, mais peuvent être plus petites, la taille étant définie par les capacités respectives de tous les intermédiaires du réseau et adaptée au cours des communications. Une application de détection doit alors être capable de repérer des fragments de documents de petite taille.

La recherche de texte dans une page web ou tout autre document est une illustration du repérage d'extraits. Les utilisateurs sont capables de retrouver tout extrait exact dans leur document source. On cherche dans ce mémoire à appliquer le même principe avec les trames passant sur le réseau : chercher le contenu des trames dans un ensemble de documents de référence.

Pour simplifier les traitements, on pourrait étudier la taille idéale des fragments à détecter et extraire uniquement ce fragment des trames reçues avant de le chercher dans les documents sources. Aux deux extrêmes, des extraits de très grande taille seraient difficilement repérables du fait du peu de chances qu'ils soient contenus en entier dans une trame, mais des extraits de toute petite taille produiraient beaucoup de faux-positifs du fait du peu d'éléments vraiment représentatifs du document original. Il faut donc soit choisir une taille de fragments assez importante pour être représentative du document et assez faible pour être contenue dans une trame, soit trouver des méthodes pour outrepasser cette question. Nous étudierons des méthodes existantes au chapitre 2.

1.2.2 Détection rapide pour filtrer efficacement

Les Fournisseurs d'Accès à Internet (FAI) gèrent les connexions des utilisateurs au réseau Internet. Ils se chargent donc de faire l'agrégation du trafic et la bande passante qu'ils utilisent est la somme de toutes les bandes passantes de leurs clients. Les FAI sont généralement reliés entre eux et aux fournisseurs de services par des fibres optiques qui autorisent des connexions à 40 Gb/s, voire 100 Gb/s. Le réseau déployé par la société Cogent [14] en est un bon exemple.

Nous avons besoin d'un point de passage des données pour pouvoir les observer et les filtrer. Les liaisons entre FAI paraissent intéressantes car il s'agit de points de passage préexistant regroupant beaucoup de trafic, et le nombre de ces liaisons est relativement limité. Il faut alors viser leur gamme de débits pour obtenir un système utilisable à grande échelle. De même, il devient impératif de traiter les paquets très rapidement puisque leur passage prend très peu de temps. En effet, les abonnés disposent couramment de bandes passantes de l'ordre de 10 Mb/s. Les paquets sont transmis à cette vitesse en 0.1 ms. Du fait des étapes par les différents éléments des réseaux pour atteindre le serveur destinataire, les utilisateurs observent en général une latence de l'ordre de quelques dizaines de millisecondes entre le départ du paquet de leur machine jusqu'à l'arrivée sur la machine visée. Une latence

trop élevée diminue le confort de l'utilisateur en le faisant patienter entre chacune de ses action. Des domaines nécessitant une bonne réactivité comme la vidéoconférence, le contrôle à distance ou les jeux vidéos en ligne sont très vite impactés par une augmentation de la latence. Il est alors important de prendre en compte cet aspect et de créer un système qui ne bloque pas, ou très peu, les paquets pendant leur analyse.

Pour augmenter les débits traités sans impacter la latence, on peut utiliser du matériel capable d'analyser les paquets en parallèle. Nous avons évoqué à la section 1.2.1 que les documents sont divisés en paquets avant d'être envoyés. Les paquets d'une source sont mélangés avec d'autres lorsque les trafics sont regroupés, notamment au niveau des routeurs. Deux paquets qui se suivent ne sont donc pas liés entre eux. De plus, la nécessité de reconnaître des fragments de documents abordée à la section 1.2.1 a été établie dans le but de travailler uniquement sur le contenu des paquets, sans devoir reconstruire le document original. Ces éléments nous offrent la possibilité de travailler sur les différents paquets en parallèle. Avec du matériel compatible, on multiplie le débit par le nombre d'unités de traitement sans augmenter la latence. Ce type de matériel est très en vogue actuellement et se développe très vite. Les plus communs sont les CPU, composés aujourd'hui de deux à douze cœurs de calcul, et les processeurs graphiques (*Graphics Processing Unit* (GPU)) qui comprennent plusieurs centaines d'unités de calcul plus limitées. D'autres dispositifs plus spécifiques existent. Les processeurs réseaux sont des puces très spécialisées dans le traitement de flux de données, utilisées principalement dans les routeurs ou cartes réseaux haut de gamme. Finalement, les *Field-Programmable Gate Array* (FPGA) présentent aussi l'intérêt d'un parallélisme intrinsèque très adaptable puisque l'on peut implémenter autant d'instances que nécessaire d'une même unité de calcul dans la seule limite de la taille de la puce. Nous nous focaliserons dans ce mémoire sur les GPU, dont nous étudierons les atouts au chapitre 2.

1.2.3 Référencement de nombreux documents

Notre système est voué à filtrer un nombre de documents quelconque, potentiellement important suivant la politique de contrôle appliquée. Il convient donc de garantir que les performances finales ne diminueront pas avec l'augmentation du nombre de documents. Les originaux à filtrer doivent être référencés, dans une base de données qui sera consultée pour identifier les données analysées. Plusieurs points entrent en compte dans les performances de cette base de données. Tout d'abord, il faut spécifier dans quel format sont enregistrés les documents référencés (ou les références à ces documents). Leur lisibilité peut être un problème puisque l'on cherche à traiter du contenu sensible. Si les administrateurs peuvent accéder à ces informations, on s'expose à des fuites de documents, ce que l'on cherche précisément à endiguer.

D'autre part, il faut tenir compte du fait que stocker beaucoup de documents peut nécessiter beaucoup de mémoire, particulièrement s'il s'agit d'images ou de vidéos. Les dispositifs de stockage disposant de beaucoup d'espace, comme les disques durs ou encore des espaces de stockage en réseau, souffrent souvent de temps d'accès assez longs. Il faudra donc chercher à réduire les besoins en espace de stockage ou le nombre d'accès nécessaires à la mémoire pour masquer la latence induite par ces accès. Là encore le parallélisme peut être utile car une application massivement parallèle a toujours des calculs en attente qui peuvent combler les temps de latence de la mémoire. La bande passante diminue alors moindrement même si la latence augmente.

Finalement, il faudra prendre en compte le problème de redondance entre les documents. En effet, des documents semblables, ne serait-ce que par l'application dans laquelle ils ont été créés ou par leur type, comporteront de nombreuses parties identiques comme les entêtes, formatages ou autres méta-données. La détection de ces parties pourra donc facilement induire des faux-positifs, c'est-à-dire des alertes lancées sur des données qui ne présentent en réalité pas de risque. Disposer d'une base de données assez importante peut permettre d'éviter ces désagréments en effectuant une analyse statistique et en ne prenant pas en compte les parties redondantes des documents lors du référencement. Nous considérerons ici que seules les parties hautement représentatives des documents et donc *a priori* uniques, sont référencées. Nous nous concentrerons ainsi sur le travail de détection et de blocage en temps réel plutôt que sur la sélection des "meilleures" parties des documents qui pourra être effectuée séparément.

1.3 Objectifs de recherche

Finalement, en reprenant les éléments précédents, nous pouvons formuler la problématique de ce mémoire comme suit :

> *Comment repérer et bloquer efficacement des contenus numériques connus transitant sur un réseau, en travaillant en temps réel avec des bandes passantes très élevées (40 à 100 Gb/s) tout en gardant une latence assez faible pour ne pas influer sur le confort des utilisateurs du réseau ?*

On proposera ainsi une solution de filtrage de contenus préalablement référencés, capable de travailler à grande échelle sur le réseau Internet ou les réseaux de grandes entreprises ou universités.

L'un des points principaux sera l'analyse des solutions parallèles qui paraissent prometteuses, ainsi que l'adjonction d'une base de données sans pertes de performances avec l'augmentation de sa taille.

1.4 Plan du mémoire

La suite de ce mémoire, s'articulera en quatre chapitres. Tout d'abord, nous étudierons au chapitre 2 les solutions existantes, tant au niveau de l'algorithme que de l'architecture du système complet. Nous ferons ainsi ressortir l'intérêt de l'algorithme de *Max-Hashing* et des systèmes basés sur des processeurs graphiques (GPU). Nous commencerons cette section par un rappel sur les protocoles utilisés dans les réseaux. Nous présenterons ensuite au chapitre 3 l'implémentation et l'optimisation de cet algorithme pour de tels processeurs, en détaillant les adaptations nécessaires pour obtenir une implémentation efficace. Nous nous intéresserons aussi aux méthodes de récupération du trafic réseau, et aux moyens simples pour filtrer ce trafic. Nous pourrons ensuite construire un banc de tests pour analyser le système au chapitre 4. Nous mesurerons les performances et répondrons ainsi à la problématique. Finalement, nous conclurons au paragraphe 5 sur l'ensemble des travaux, sur les limitations et les améliorations restantes. Nous signalerons aussi les problèmes de neutralité qui existent sur un tel système.

Le lecteur trouvera en annexe le code annoté décrivant l'analyse les données reçues du réseau pour y rechercher les correspondances avec les documents originaux.

CHAPITRE 2

REVUE DE LITTÉRATURE

Le but de ce mémoire est de présenter un système fonctionnel de filtrage de contenus numériques connus sur un flux réseau. Pour ce faire nous allons d'abord étudier les algorithmes de reconnaissance existants qui pourraient être adaptés à ce domaine, avant de nous intéresser aux méthodes d'analyse de flux réseaux. Nous insisterons sur les différents types de matériel communément utilisés et nous pourrons analyser les forces et les faiblesses de chacune des approches pour finalement constituer la meilleure réponse possible à notre problème.

2.1 Protocoles réseaux

Comme nous l'avons évoqué en introduction, les réseaux utilisent en général le protocole Ethernet. D'autres protocoles s'ajoutent dans celui-ci pour masquer les adresses matérielles et abstraire les communications. Nous allons détailler ici leur fonctionnement car nous serons amenés à les utiliser tout au long de ce mémoire. Le lecteur déjà à l'aise avec les protocoles Ethernet, IP, TCP et UDP pourra toutefois passer cette section sans incidence sur la suite de la lecture.

2.1.1 Connexion Ethernet

Ethernet est un protocole de communication au niveau physique normalisé en 1985 [33]. On construit les messages en commençant par un en-tête identifiant l'émetteur et le récepteur, avant de placer le contenu (ou *payload*). L'ensemble transmis est appelé une "trame". La taille du contenu est limitée à 1500 octets. Si les informations à transmettre sont trop volumineuses, alors elles sont divisées en plusieurs paquets de taille inférieure ou égale à 1500 octets. Le destinataire se charge de reconstruire l'information complète après avoir reçu l'ensemble des paquets correspondants.

Chaque machine est identifiée par une adresse unique, appelée adresse *Media Access*

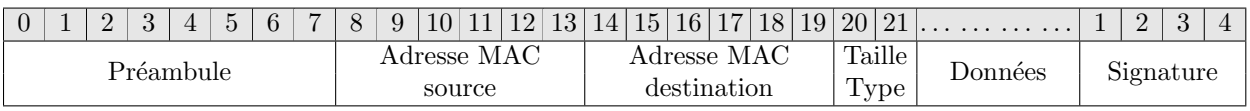


Figure 2.1 – Modèle de paquet Ethernet II

Control (MAC), constituée de six octets. Il y a plusieurs types de trames Ethernet, avec chacune leur en-tête spécifique. La plus utilisée est Ethernet II. Son en-tête est très simple, comme l'indique la figure 2.1 :

- Adresse MAC de la machine destinataire
- Adresse MAC de la machine source
- Type/Taille : détermine quel type de données est contenu dans la trame :
  → valeur ≤ 1500 : il s'agit directement de données brutes de la taille correspondante
  → valeur > 1500 : il s'agit du type de protocole utilisé dans le champ données
- Données, jusqu'à 1500 octets
- Signature pour vérifier l'intégrité des données reçues

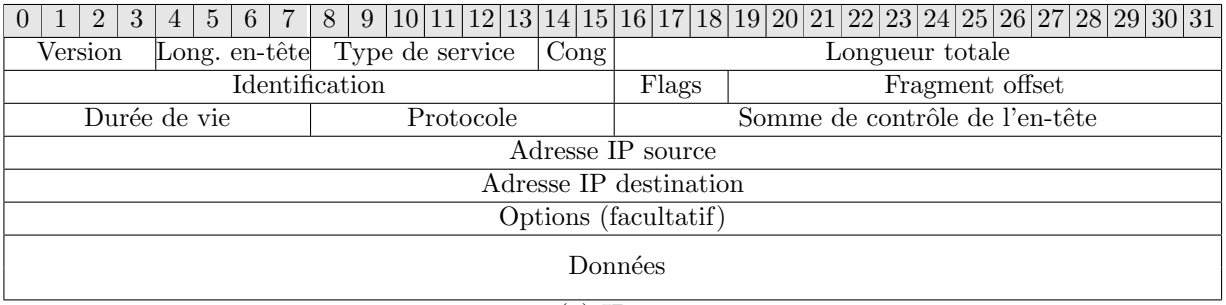
Il ne transite donc sur les réseaux que des trames de ce modèle ou équivalent. Il suffit alors de pouvoir les récupérer pour accéder à leur contenu. Il n'y a aucun contrôle ni restriction quant à la lecture par une autre entité que le destinataire. Les deux machines concernées ne sont pas nécessairement reliées directement l'une à l'autre et le message peut être retransmis par plusieurs intermédiaires, qui ont alors pleinement accès au contenu.

2.1.2 Protocole TCP/IP

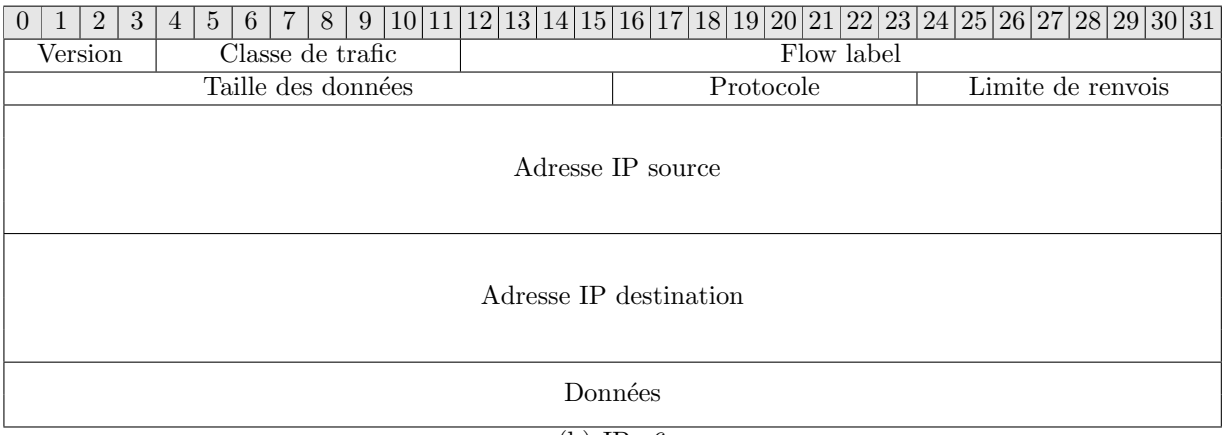
On l'a vu, un protocole peut être indiqué dans les entêtes des trames Ethernet. Les données sont en effet très rarement transmises directement. Des abstractions sont ajoutées au dessus de l'Ethernet pour faciliter le routage des trames. La plus courante est l'*Internet Protocol* (IP). Il s'agit d'insérer un nouvel en-tête dans les données de la trame Ethernet. On communique alors entre machines non plus avec l'adresse MAC, considérée comme une adresse matérielle, mais avec une adresse logicielle, l'adresse IP. Plusieurs contrôles existent avec ce protocole comme la recherche de destinataires avec le DNS pour retrouver l'adresse IP à partir du nom de la machine ou l'ARP pour en retrouver l'adresse MAC à partir de l'adresse IP, ou encore l'assignation dynamique d'adresse avec le DHCP, qui permettent d'étendre les possibilités de communication. La version 4 de L'IP utilise 32 bits pour coder les adresses, la version 6 en cours de déploiement utilise dorénavant 128 bits. L'augmentation de la taille des adresses permet d'autoriser la connexion simultanée de beaucoup plus de périphériques dans un même réseau.

L'en-tête, schématisé à la figure 2.2, reprend le même principe que l'en-tête Ethernet, avec de nouveaux paramètres. Les paramètres communs aux deux versions sont les plus utiles :

- Version : spécifie quelle version de l'en-tête IP est utilisée
- Taille
- Protocole du contenu

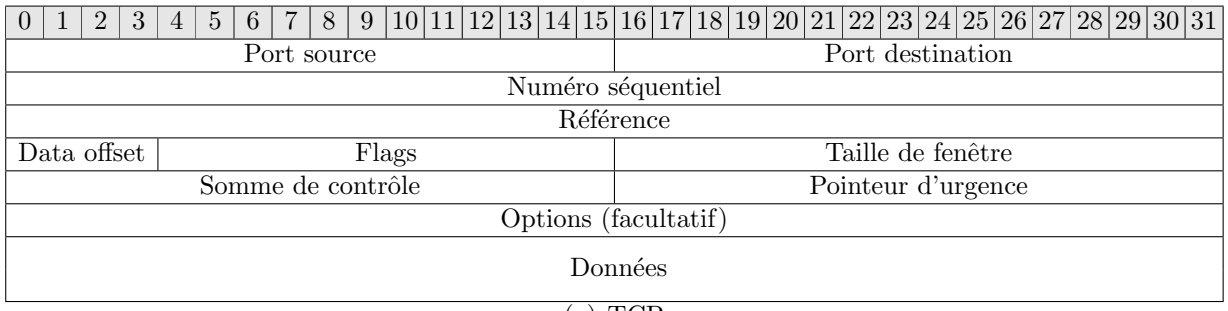


(a) IP v4

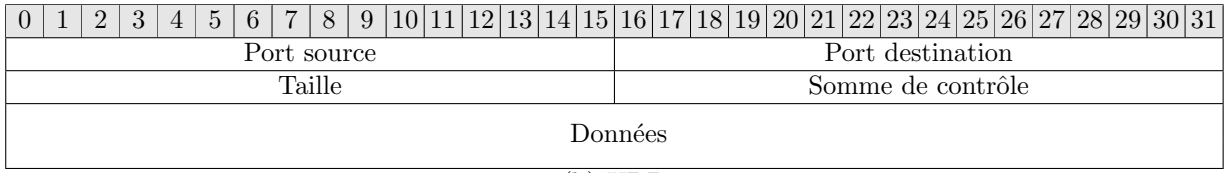


(b) IP v6

Figure 2.2 – Modèle de paquet IP



(a) TCP



(b) UDP

Figure 2.3 – Modèle de paquet TCP et UDP

– Adresses IP de l'émetteur et du destinataire

Lors de transferts de données entre deux machines, on utilise en général une abstraction supplémentaire, dite couche de transport, telle que le *Transmission Control Protocol* (TCP) ou l'*User Datagram Protocol* (UDP). Le TCP comprend des accusés de réception et est donc utilisé dans la majorité des cas, dès lors que l'on veut s'assurer que les données arrivent toutes à destination L'UDP, qui n'a aucun contrôle de ce type, est préféré dans la transmission de flux en temps réel. L'un comme l'autre utilisent l'adressage du protocole IP et ajoutent des "ports" aux machines, qui peuvent être considérés comme des accès indépendants, chacun dédié à une application logicielle sur ces machines. On adresse ainsi chaque paquet à un service donné sur la machine destinataire et on isole les services les uns des autres. Comme indiqué sur la figure 2.3, le port est identifié par un numéro compris entre 0 et 65535 La plupart des identifiants inférieurs a 1024 sont standardisés [31] et réservés à certains services courants. La connexion aux serveurs web pour naviguer sur un site Internet se fait par exemple usuellement par le port 80.

On arrive alors à un niveau où des programmes s'adressent aux programmes d'autres machines, grâce au couple adresse IP et port. Les abstractions supplémentaires qui existent sont des couches applicatives que nous ne considèrerons donc pas puisque nous n'avons pas besoin de nous spécialiser sur l'écoute des communications d'un programme spécifique. Une connexion est alors définie par les quatre paramètres suivants :

– Adresse IP de l'émetteur
– Port de l'émetteur
– Adresse IP du destinataire
– Port du destinataire

2.2 Algorithmes de détection de données connues

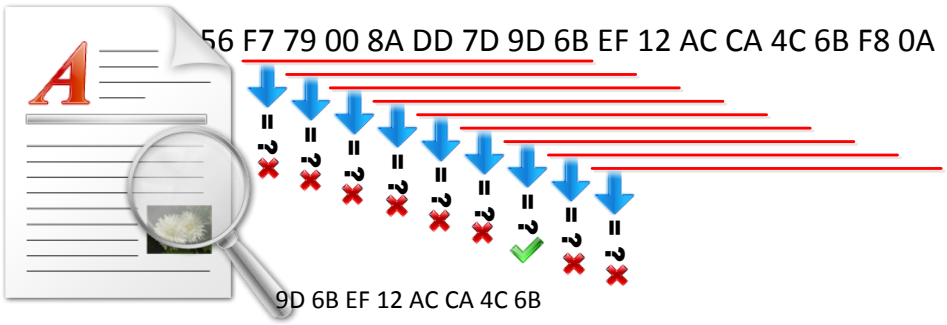
Reconnaître des motifs (en anglais "*Pattern recognition*") dans un flux de données est à la base de nombreux algorithmes, depuis la détection de plagiat jusqu'aux antivirus. Il s'agit de balayer un texte, un fichier, ou n'importe quel ensemble de données pour y trouver un motif ou *pattern*, ou l'ensemble de ses occurrences. Nous cherchons ici à analyser une connexion réseau. Le protocole le plus répandu est l'IP, lui-même intégré dans le protocole Ethernet. Comme nous l'avons évoqué à la section 2.1, les données transférées selon ces protocoles sont divisées en paquets. On peut facilement "écouter" une connexion réseau et récupérer les paquets qui y transitent pour analyser leur contenu. Nous présenterons ces méthodes dans le chapitre suivant. Il paraît alors intéressant d'appliquer les méthodes de reconnaissance de documents au contenu des paquets IP pour détecter le passage de données connues (interdites, illégales, secrètes ou autres) sur un flux réseau. Il suffirait alors de comparer ce contenu avec une liste de fichiers à repérer et de lancer des alertes en cas de correspondance.

À partir de ces idées, on peut établir un premier cahier des charges pour caractériser simplement l'algorithme de détection des données :

- Possibilité de repérer des fragments de documents et pas uniquement des documents complets, pour travailler avec le contenu de chaque paquet individuellement.
- Référencer un nombre important et quelconque de documents, quels que soient leur type et leur taille, sans perte de performance.
- Bande passante très importante pour pouvoir travailler à l'échelle de grandes entreprises ou de FAI.

2.2.1 Reconnaissance de chaînes de caractères

Les chaînes de caractères sont très simples à représenter et à traiter, d'autant plus que nous pouvons facilement les appréhender. Chaque caractère est représenté par un octet, et tout ensemble de données peut alors être considéré comme une chaîne de caractères pas nécessairement lisible, il suffit pour s'en convaincre d'ouvrir un fichier quelconque avec un éditeur de texte. Il n'est donc pas surprenant que celles-ci aient été à la base des recherches dans le domaine. Le problème consiste à localiser une chaîne de caractères ou toutes ses occurrences dans un texte. On trouve notamment deux études sur les évolutions de tels algorithmes par Baeza-Yates [3] et Michailidis et Margaritis [50].



Chaque octet est comparé avec le modèle recherché.

Figure 2.4 – Principe de comparaison directe

Comparaison directe

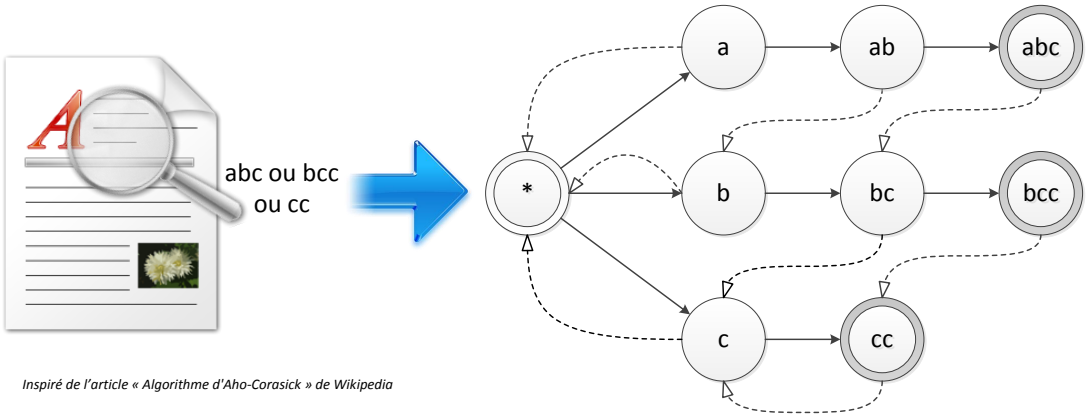
La première méthode adoptée a été une simple comparaison de la chaîne de caractères recherchée avec chaque partie du texte analysé. Cette approche a été utilisée dès les débuts de l'informatique mais n'a été étudiée rigoureusement que plus tard par Barth [4]. Elle est très simple, puisqu'il s'agit de déplacer le modèle recherché tout au long des données étudiées en comparant tous les caractères à chaque itération. Elle se classe donc parmi les méthodes de force brute, avec un temps d'exécution proportionnel à la taille du texte ainsi qu'à celle de la recherche. La figure 2.4 décrit le principe de base.

Pour réduire le nombre de comparaisons, différents algorithmes ont été proposés, optimisant le déplacement du modèle au long du texte analysé. Ceux de Boyer et Moore [6] et Knuth *et al.* [41] sont devenus les références dans le domaine, et furent ensuite améliorés par Horspool [28] ou encore Takaoka [68]. Ces approches ajoutent un traitement préalable visant à réduire le nombre d'itérations, par exemple en repérant des parties redondantes. On obtient par ce biais des méthodes de moindre complexité par rapport à la longueur du texte et/ou du modèle.

Expressions régulières

Pour réduire l'encombrement et la diversité des chaînes de caractères recherchées, des méthodes de représentation de ces chaînes de caractères ont été présentées. Il devient par exemple possible de spécifier qu'un caractère peut être répété, que n'importe quel caractère peut être placé à certains emplacements, etc. pour finalement décrire beaucoup plus largement des recherches qui peuvent s'adapter à différents cas. Une explication très complète est donnée par Aho [1].

À l'origine, cette approche n'était pas dédiée à la comparaison de documents. En effet, McCulloch et Pitts [47], puis Kleene [40] ont érigé les fondements théoriques dans le domaine



Une machine à états est créée à partir du modèle recherché. La lecture des données fait évoluer l'état courant jusqu'à aboutir à un état final.

Figure 2.5 – Machines à états de l'algorithme d'Aho-Corasick

des automates, en se basant sur des études et simulations de neurones. Leurs neurones formels pouvaient être excités ou inhibés, et produisaient une sortie en conséquence. Plusieurs évolutions et simplifications ont ensuite mené aux travaux de Rabin et Scott [60], qui ont repris, élargi et prouvé les théories précédentes. Plusieurs méthodes de traitement ont été proposées, basées sur un principe général commun : créer des machines à états pour représenter l'ensemble des objets recherchés. On lit ensuite les données étudiées, et les caractères rencontrés activent certaines transitions des machines à états, dont l'état courant évolue au fur et à mesure de la lecture. Lorsque celles-ci rencontrent un état final, alors un *pattern* recherché a été trouvé.

Les algorithmes sont souvent classés en deux catégories :

- *Deterministic Finite Automaton* (DFA) [47] : La méthode d'origine dans laquelle les états représentent les avancées caractère par caractère. On ne peut alors évoluer dans la machine que d'un état à la fois, à la lecture de chaque caractère, avec des transitions exclusives. Le système n'est constitué que d'une machine à état.
- *Non-Deterministic Finite Automaton* (NFA) [60] : Théorie plus générale selon laquelle les états sont beaucoup plus ouverts, on peut effectuer une ou plusieurs transitions à chaque lecture, voire même sans lire de caractère. De plus, plusieurs transitions peuvent être activées simultanément à la sortie d'un état et plusieurs petites machines peuvent être exécutées simultanément.

L'implémentation des automates a été très largement étudiée par la suite. Aho et Corasick ont proposé un arbre de caractères [2] qui est devenu l'approche classique car très simple et efficace. Chaque état représente un caractère des motifs recherchés, mutualisant les préfixes identiques pour réduire le nombre de branches. La figure 2.5 présente ce principe. Plusieurs

évolutions se sont basées sur ces travaux pour ajouter certaines propriétés spécifiques. On citera par exemple des implémentations comme celle de Liu *et al.* [44], autorisant des recherches à très haute vitesse sur GPU (de l'ordre de 80 à 100 Gb/s) en divisant le texte étudié en segments de la longueur maximale d'un motif, tout en optimisant la machine à états et l'algorithme pour les spécificités des processeurs graphiques.

Les expressions régulières permettent de limiter le stockage nécessaire tout en offrant des recherches beaucoup plus générales puisque l'on remplace la multitude d'éléments recherchés par un modèle unique. On simplifie alors la maintenance mais on ajoute un processus nécessaire de vérification pour s'assurer de l'exhaustivité du modèle.

Le traitement par machine à états requiert que les modèles soient assez courts et peu nombreux (quelques milliers de modèles d'une centaine de caractères), sans quoi les calculs pour résoudre ces machines à états deviennent très complexes, longs, voire impossibles. De plus, dès lors que les *patterns* utilisés sont de petite taille pour être efficaces, il est difficile de les adapter à la détection de fichiers plus volumineux tels que des rapports ou des images.

Reconnaissance approximative

Il existe un grand nombre de méthodes pour comparer des chaînes de caractères et détecter les ressemblances en autorisant un certain nombre d'erreurs, qu'on appelle "Approximate String Matching". Elles sont très utilisées pour détecter les fautes d'orthographe ou pour rechercher des fragments d'ADN avec la possibilité d'une mutation. L'idée est de calculer le nombre de différences entre deux chaînes de caractères (modification, suppression ou ajout d'un caractère). On compare ainsi un modèle avec tous les originaux puis on compare les résultats pour trouver les plus proches.

De nombreux algorithmes implémentant des distances différentes ont été proposés. Une revue très complète a été publiée par Navarro [52], retraçant plus en détail les possibilités offertes dans ce domaine. Cette approche est aussi souvent utilisée pour de la recherche de plagiat ou de la copie illicite de documents [18].

Stockage de chaînes de caractères

Rechercher des chaînes de caractères, ou des modèles de chaînes demande de stocker ces modèles, qui sont des chaînes de caractères dont la longueur est indéterminée. Leur stockage peut donc poser plusieurs problèmes. Tout d'abord, rechercher un document reviendrait à stocker la quasi-totalité de ce document dans une nouvelle base de données, ce qui peut rapidement demander un espace de stockage gigantesque lorsque le nombre de documents à repérer devient important. Il faut alors utiliser des stockages de plus grande taille (disques

durs, espaces de stockage en réseau, etc.) qui sont souvent beaucoup plus lents. Les performances du système complet sont alors rapidement limitées par la faible bande passante et surtout les latences importantes des accès à la mémoire.

D'autre part, stocker des données hétérogènes mène souvent à des implémentations de bases de données moins efficaces. Les opérateurs de comparaison par exemple, doivent connaître la taille de la plus grande valeur stockée afin de comparer les autres sur la même base, qui revient à effectuer des comparaisons sur des données de très grande taille, ralentissant ainsi leur fonctionnement. On préfère souvent indexer des données de taille fixe, idéalement assez limitée, pour que les processeurs puissent les traiter efficacement.

La sensibilité des documents à stocker est elle-aussi un point important. À partir du moment où des données sensibles (interdites ou confidentielles par exemple) sont regroupées, il convient de les entourer de beaucoup plus de sécurité contre les vols ou les accès non autorisés. La constitution de telles bases de données peut même être interdite dans certains cas.

L'utilisation directe des chaînes de caractères constitue donc un inconvénient et une limitation majeure. Il apparaît dès lors plus intéressant d'utiliser des représentations illisibles des données, codées et ne pouvant être décodées.

2.2.2 Méthodes de Hachage

Pour accélérer les comparaisons de documents et simplifier la base de données de référence, on utilise souvent des signatures ("*hashes*"), aussi appelées empreintes ou clés, qui sont calculées à partir du contenu du fichier ou de l'ensemble de données traité. Dans la majorité des cas, on crée une signature de taille fixe, quelle que soit la taille des données sources. La principale propriété recherchée est l'injectivité : si deux empreintes sont identiques alors la source qu'ils représentent doit être la même. Ceci n'est vrai qu'avec une certaine probabilité étant donné le nombre fini de valeurs différentes de taille fixe. Par exemple, la probabilité que deux mots aléatoires codés sur 64 bits soient identiques est 2^{-64}. Malgré tout, la grande majorité des applications utilisant le hachage se base sur cette propriété, avec des signatures de plus grande taille pour limiter la probabilité de collision, c'est-à-dire que deux sources différentes soient représentées par la même signature.

On l'a vu, sauvegarder des signatures à la place des données d'origine présente plusieurs intérêts. Cela permet tout d'abord de ne pas travailler avec du contenu potentiellement confidentiel et/ou privé mais avec des représentations plus ou moins déchiffrables de ce contenu. Ensuite, cette approche rend les traitements beaucoup plus efficaces car le type des données est connu et uniforme. On peut par exemple définir les signatures comme des entiers positifs codés sur 128 bits et ainsi optimiser la recherche et l'indexation pour ce type

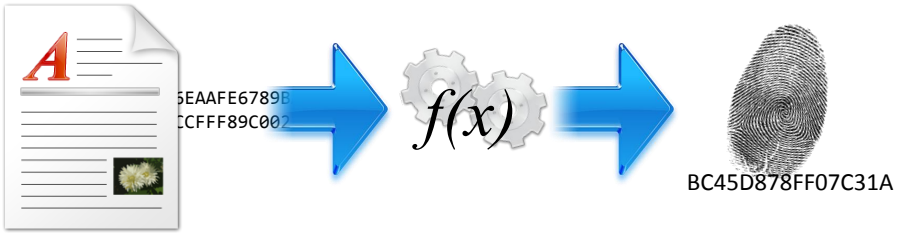
de données. Finalement, le stockage est là encore facilité car tous les éléments sont d'une taille fixe (ou au moins connue et bornée), ce qui permet une meilleure utilisation de l'espace mémoire, pour finalement référencer un nombre quasi-illimité de documents.

Le domaine judiciaire, auquel peut s'apparenter notre système, utilise largement le hachage pour les recherches de preuves parmi de grandes quantités de données ou l'analyse en temps réel de communications pour prévenir les menaces. Roussev a beaucoup travaillé sur ce type d'application dans ses articles [63, 64, 65]. L'auteur cite notamment des enquêtes impliquant d'immenses quantités de données à analyser, comme une saisie de 60 To de documents à propos de la guerre en Irak [61]. Calculer les signatures de fichiers permet d'en écarter rapidement les plus communs, tels que les fichiers des systèmes d'exploitation ou de programmes répandus. Le NIST (National Institute of Standards and Technology) [51] maintient pour cela une base de données qui contient les signatures de tels fichiers. Les enquêteurs peuvent la consulter afin de trier rapidement le contenu de disques durs saisis et de concentrer leurs recherches sur les fichiers les plus intéressants.

Méthode de base

La méthode de base, telle qu'on l'utilise dans les exemples précédents, consiste à calculer les signatures de chacun des documents recherchés, puis celles des documents analysés et de les comparer. La figure 2.6 résume le principe. Lorsque les signatures sont identiques, on peut affirmer que les documents sont les mêmes selon la probabilité exposée dans le paragraphe précédent. Ce principe est souvent utilisé sur Internet pour télécharger de gros fichiers, avec lesquels on fournit la signature afin que l'utilisateur puisse vérifier que son fichier n'a pas été corrompu lors du transfert. Les algorithmes les plus utilisés sont *Message Digest #5* (MD5) et *Secure Hash Algorithm* (SHA1).

Une première limitation intervient dans le traitement des documents : la moindre modification implique une valeur de signature différente. Si cette propriété est utilisée pour vérifier



Méthode de base : on calcule une empreinte par fichier, à partir de l'ensemble du fichier.

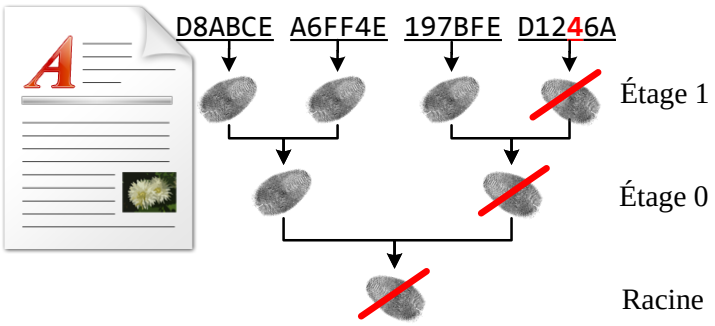
Figure 2.6 – Principe du hachage

l'intégrité d'un ensemble de données, elle devient une contrainte lorsque l'on veut comparer des documents proches, ou légèrement corrompus. Des changements seront invisibles pour l'utilisateur, par exemple un octet ajouté à la fin du fichier, mais sa signature sera différente et il ne pourra donc pas être repéré. Plus important pour notre étude, on ne peut pas retrouver des fragments de documents avec cette approche mais uniquement les documents complets. Or, comme on l'a annoncé au chapitre précédent, nous devons travailler sur les paquets réseaux, qui sont précisément des fragments de ces documents. Il n'est pas possible de reconstituer les originaux envoyés et nous devons donc raffiner cette méthode.

Fragmentation

Une première réponse au problème de fragmentation des données a été proposée par Merkle [49] avec son "*hash tree*" aussi appelé "arbre de Merkle". Le principe consiste à diviser la source en plusieurs blocs de taille constante avant de calculer l'empreinte de chacun de ces blocs. On crée ensuite plusieurs étage en calculant à chaque fois une nouvelle empreinte à partir de pairs de l'étage précédent. On divise ainsi par deux le nombre de valeurs de chaque étage jusqu'à arriver à la racine de l'arbre.

Cette méthode permet de vérifier l'intégrité de documents de manière efficace. En effet, lorsqu'une partie du document est altérée, seule la signature correspondant au bloc concerné est modifiée, ainsi que toutes les clés des étages supérieurs qui mènent à celle-ci. On n'a alors plus qu'à redescendre l'arbre depuis la racine vers les feuilles pour trouver le bloc mis en cause, ce qui réduit le nombre de comparaisons nécessaires. Cette méthode est utilisée pour transférer des ensembles importants de données, pour n'avoir qu'à renvoyer la partie altérée au lieu de l'ensemble de la source. La figure 2.7 illustre le principe.



On divise cette fois le fichier en blocs de taille fixe et on calcule ensuite une signature pour chacun d'eux, puis on crée un arbre de signatures. Une erreur est facilement repérable dans la source, il suffit de remonter l'arbre.

Figure 2.7 – Principe du *Tree Hash*

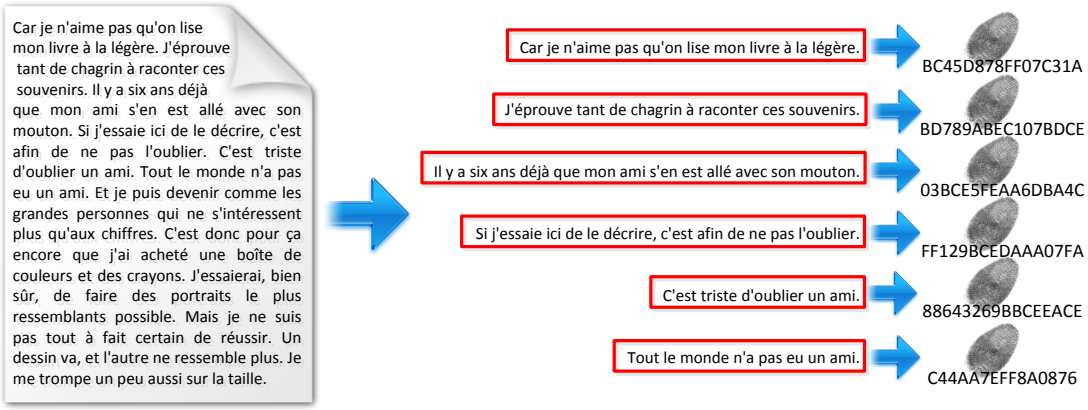
Pourtant, cette méthode ne peut s'appliquer facilement aux paquets transmis sur un réseau du fait de la taille variable de ceux-ci et de la division imprévisible lors de la création des paquets. On ne peut prévoir ni le fractionnement ni la configuration des blocs qui seront transmis. Il paraît impossible de spécifier une taille fixe qui garantirait que ces blocs soient transmis alignés correctement dans les paquets sur le réseau. Il faut donc trouver une méthode plus précise, qui ne se contentera pas de diviser un document en blocs fixes.

Contexte

Dans la méthode précédente, la moindre suppression ou insertion de données dans un fichier modifie tous les blocs subséquents ainsi que toutes les signatures correspondantes. Par exemple, si un octet est inséré au début d'un document, toutes les signatures le caractérisant seront modifiées car le contenu de chaque bloc sera décalé. Le document modifié sera donc considéré comme un document entièrement nouveau, sans aucun lien avec l'original. La même idée s'applique pour les paquets Internet, puisqu'il s'agit de travailler sur un extrait quelconque (en taille et emplacement) du document, les blocs n'ont que très peu de chances de se trouver placés correctement dans des paquets Internet pour être détectés.

Les premiers à avoir proposé une solution à ces défis sont Hunt et McIlroy [29], qui étudiaient les textes par ligne, en recherchant la plus longue suite de correspondances. Tridgell a présenté en 2002 un logiciel appelé "SpamSum" [71] permettant de détecter les spams, textes généralement semblables les uns aux autres sans être réellement identiques. Sa méthode, proche du travail de Rabin [59], consiste à calculer des signatures ne dépendant que d'une petite fenêtre de calcul glissant sur l'ensemble des données. Cette idée sera reprise dans le domaine de la sécurité par Kornblum qui la baptisera "*Context-Triggered Piecewise Hashing* (CTPH)" [42], puis améliorée par Long et Guoyin en 2008.

Cette méthode développe la précédente : le document est divisé en blocs et les signatures sont calculées sur ces blocs uniquement. Ici les blocs sont beaucoup plus petits, on parle plutôt de contextes ou de fenêtres. Leur taille n'est pas constante et toutes les fenêtres ne donnent pas nécessairement lieu au calcul d'une signature. Tout revient à déterminer l'emplacement de ces fenêtres dans un ensemble de données, puis à déterminer quelles signatures seront conservées, à l'aide de propriétés et paramètres facilement réutilisables et applicables partout. Un premier exemple simple consisterait à garder l'idée de blocs de taille fixe, mais de très petite taille, en ne conservant que les signatures vérifiant une propriété donnée (par exemple un résultat modulo un paramètre). De manière plus élaborée et en illustrant l'idée de blocs de taille ne pouvant être prédéterminée, on pourrait décider de calculer une signature par phrase dans un texte. Ce principe est illustré sur la figure 2.8.



Les repères sont d'abord calculés afin de diviser le fichiers en petits contextes (ici par phrase du texte). On calcule alors une signature par contexte.

Figure 2.8 – Principe du *Context-Triggered Piecewise Hashing*

Dès lors que ces fenêtres ne dépendent que de propriétés locales et non plus du document au complet (exemple : une phrase), elles peuvent être facilement détectées lorsqu'elles sont déplacées. On peut donc combattre le plagiat beaucoup plus efficacement, mais aussi repérer une fenêtre dans un bloc de données quelconque à partir du moment où celle-ci est contenue en entier dans ce bloc, ce qui devient possible si leur taille est assez faible. La détection de données connues par analyse du contenu de paquets réseaux est donc rendue possible puisque l'on n'a plus besoin de considérer le document original dans son ensemble.

Si cette méthode est très robuste contre la fragmentation, elle demande malgré tout un espace de stockage important afin de pouvoir enregistrer le plus de signatures possible. En effet, chaque signature ne dépend plus que d'un petit segment dans un grand ensemble de données, et on ne représente réellement cet ensemble qu'en gardant plusieurs signatures. Le repérage d'un fichier revient alors à détecter un certain nombre de signatures appartenant au même fichier. On peut choisir une limite minimale en nombre de signatures détectées avant de lancer des alertes de détection pour valider statistiquement le repérage.

D'autre part, les propriétés utilisées pour la spécification des fenêtres dans le document ne peuvent garantir qu'un nombre suffisant de fenêtres sera créé pour représenter fidèlement un document, ni que leur répartition sera régulière sur l'ensemble de ce même document. À l'extrême, on ne peut même pas garantir que la propriété recherchée existera dans le document. Ainsi, cette méthode paraît efficace par son principe, mais assez aléatoire dans son application puisque le nombre de signatures par document reste variable.

D'autres méthodes du même type se basent sur le principe de calculer les signatures de petits blocs dépendant d'un voisinage limité dans les documents. Garfinkel *et al.* insistent sur l'utilité de ces méthodes pour rechercher des fragments de données connues, notamment

dans les paquets IP [25]. Les tests qu'ils effectuent sur des images et des vidéos utilisent en particulier la fragmentation en blocs du disque dur. Les auteurs ajoutent aussi une analyse statistique des signatures pour supprimer les redondances. Ils montrent ainsi une très bonne précision de repérage et nous poussent donc à continuer dans ce sens.

Accélération des comparaisons

Les comparaisons sont plus efficaces lorsque l'on utilise les empreintes à la place des fichiers. Leur nombre reste par contre important, d'autant plus si plusieurs signatures doivent être enregistrées pour chaque fichier. Lorsque l'on analyse un document ou un ensemble de données, il faut comparer chacune des signatures calculées avec les signatures de tous les fichiers originaux, ce qui requiert un grand nombre d'accès à la mémoire qui les contient. Il devient alors primordial d'optimiser ces accès pour limiter l'accumulation de latences qui impactent directement les performances du système.

Les approches classiques utilisent des bases de données sous forme de tableaux triés. On dispose alors d'algorithmes de recherche évolués pour limiter le nombre de lectures nécessaires avant de trouver une valeur. Le nombre d'accès plus faible reste cependant proportionnel à la taille de la base de données. De plus, lorsque le nombre de documents référencés augmente, on doit stocker la base de données sur des médias de taille plus importante, qui sont aussi souvent plus lents, augmentant ainsi les latences. Beaucoup d'algorithmes utilisent le filtre de Bloom [5] pour accélérer les recherches dans les bases de données [7]. Il s'agit d'un intermédiaire pour les recherches qui crée une réplique résumée de la base de données pouvant alors être stockée sur des mémoires plus petites et plus rapides. Les filtres de Bloom peuvent produire des faux-positifs, mais jamais de faux-négatifs. On peut donc les utiliser pour s'assurer de l'absence d'une signature dans la base de données, mais on doit en confirmer la présence en accédant cette fois à la base de données complète.

Algorithme de winnowing

Pour supprimer les incertitudes dans la répartition et le nombre de signatures calculées pour chaque document, certaines méthodes reprennent l'idée du CTPH en considérant de petites fenêtres dans un bloc de données et leurs signatures, mais assurent de conserver une signature ou plus. L'algorithme *winnowing* [66], dont le principe est illustré sur la figure 2.9, calcule par exemple les signatures de tous les ensemble de k octets consécutifs, puis considère tous les blocs de n signatures consécutives. Dans chaque bloc il sélectionne la signature de valeur minimale. Lorsque plusieurs blocs ont la même valeur minimale, on note l'emplacement de la signature le plus à droite parmi ces blocs et on ne conserve dans

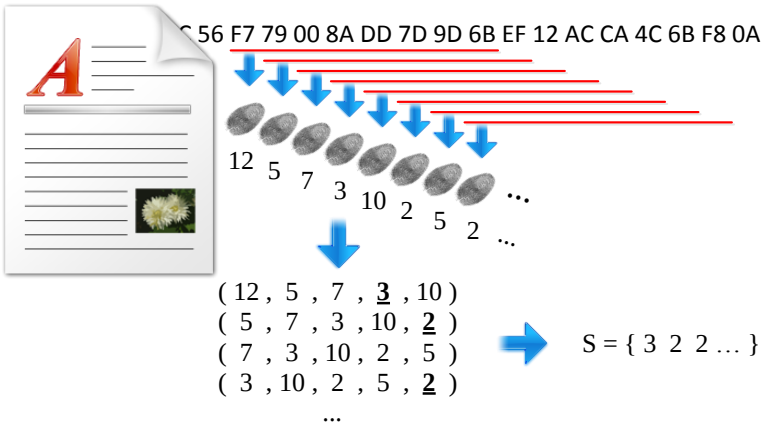


Figure 2.9 – Principe du *winnowing*

la sélection que les signatures qui sont effectivement à cet emplacement. Finalement, on concatène l'ensemble des valeurs sélectionnées comme représentant du bloc de données. On ne connait donc pas la taille de la signature à l'avance.

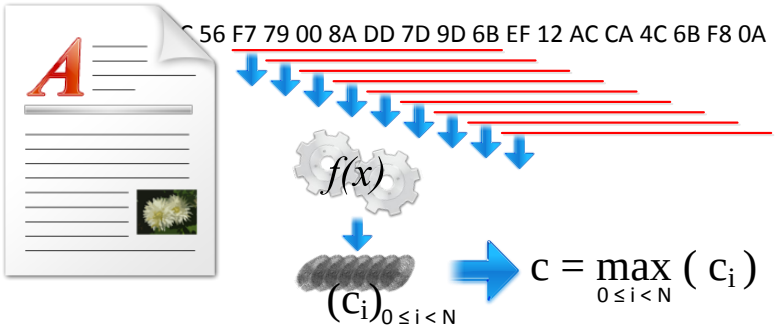
2.2.3 Max-Hashing

L'algorithme de "*Max-Hashing*" utilise un principe semblable à celui du *winnowing* mais plus simple : la signature finale est simplement celle qui présente la valeur maximale. De plus, le document original est préalablement découpé en blocs sur lesquels sont calculées les signatures. La méthode, résumée sur la figure 2.10, est relativement simple à mettre en place. Il suffit de fixer quatre paramètres :

- une taille de fenêtre F,
- une taille de bloc B pour diviser le document source, pouvant être défini à partir du nombre de signatures voulu pour le document,
- une fonction de calcul des signatures `f` applicable aux fenêtres,
- une fonction de maximum `max` applicable sur l'ensemble des signatures calculées.

On fait alors avancer la fenêtre octet par octet sur l'ensemble du document, et l'on calcule une signature de la fenêtre à chaque itération. Pour un document de taille D on obtient donc $N = D - F + 1$ signatures. La méthode de base consiste ensuite à les diviser en blocs de B signatures successives. Finalement, pour chaque groupe, on garde la signature ayant la valeur maximale. On obtient alors $\lceil \frac{D-F+1}{B} \rceil$ signatures pour représenter le document, réparties de façon homogène.

On garde ainsi l'aspect représentatif de la signature, puisque celle-ci dépend de l'ensemble des données par le calcul du maximum, mais aussi la précision de détection, puisque les signatures calculées ne se basent que sur de petites fenêtres qui n'ont que très peu de chances



La fenêtre de calcul parcourt l'ensemble des données. Les signatures sont calculées à chaque pas, et la signature de valeur maximale est conservée.

Figure 2.10 – Principe du *Max-Hashing*

d'être coupées ou altérées. Dès lors que l'on analyse un bloc de données qui contient cette fenêtre, elle sera repérée car sa signature aura toujours la valeur maximale comparée à celles de son voisinage direct. On détecte donc la présence du document à partir du moment où l'on analyse le petit fragment contenant un voisinage relativement restreint autour de cette fenêtre. Les risques de segmentation de l'envoi par le réseau sont très réduits car on peut utiliser des fenêtres très courtes.

Cet algorithme a été développé en 2008 et une demande de brevet a été déposée [15]. Les méthodes utilisées pour le calcul et le choix des signatures seront développées à la section 3.3. Nous utiliserons et adapterons cet algorithme pour qu'il réponde à nos besoins. Une version logicielle et une version matérielle sur FPGA ont été implémentées. Nous allons étudier dans la section suivante les divers types de matériels qui pourraient être adaptés, pour finalement montrer les avantages potentiels à utiliser des GPU.

2.3 Systèmes d'analyse de flux réseaux

La section précédente étudiait les algorithmes permettant de détecter des données dans des paquets IP, cette section va maintenant s'intéresser aux implémentations existantes de tels systèmes, afin de déterminer une architecture optimale pour détecter et bloquer les données. L'implémentation n'est en effet pas triviale car les connexions réseaux sont aujourd'hui très rapides (de l'ordre de plusieurs dizaines ou centaines de gigabits par seconde) et les débits continuent bien entendu leur croissance. Les FAI doivent traiter ces données et les rediriger vers les bons fournisseurs de services sans ralentir le trafic, pour que le client aie finalement un accès stable à très haut débit, quel que soit le service auquel il se connecte. À ce niveau, le regroupement du trafic des nombreux utilisateurs conduit déjà à des débits

de l'ordre de 40 à 100 Gb/s et il faut donc disposer de systèmes très puissants si l'on veut pouvoir analyser le trafic en temps réel à de telles vitesses.

Le SANS Institute [1], regroupement d'experts en sécurité informatique, a publié un document [23] qui résume les principales démarches à suivre dans la surveillance de réseaux informatiques. Deux parties peuvent être distinguées. La première est l'analyse *a posteriori*, qui consiste à retracer les événements après un incident et finalement comprendre ce qui a conduit à cet événement pour être en mesure de corriger des failles et éviter de futurs problèmes. La seconde partie de la surveillance est exécutée en temps réel, il s'agit de la prévention active : en analysant certaines métriques du réseau, et en les comparant aux scénarios connus d'attaques ou d'autres problèmes, il est possible de détecter du trafic suspect et ainsi de prendre des mesures rapidement pour l'empêcher de nuire. Nous étudierons dans une première partie l'analyse *offline* pour en connaître les forces et des limites, puis nous nous pencherons plus précisément sur les méthodes de détection en temps réel, car notre système s'y apparente clairement.

2.3.1 Retour sur incident

Un des problèmes récurrents des administrateurs est la difficulté de retracer, même sommairement, les événements qui ont conduit à l'apparition d'un incident sur le réseau. Ils doivent ainsi inspecter les *logs* des différents composants de leur réseau (firewall, proxy, etc.) dans l'espoir d'y remarquer des anomalies. Les chances deviennent très faibles avec l'augmentation des bandes passantes. La rapidité des transferts actuels conduit en effet à la production de gigantesques quantités de données en très peu de temps.

Stockage des logs

L'analyse se base sur le trafic passé. Il convient de l'avoir stocké et d'être apte à l'étudier. La première étape consiste en effet à être capable de copier et de stocker durablement toute l'activité observée sur un réseau. Tous les composants d'un réseau tels que la passerelle Internet, le pare-feu ou le serveur de stockage, enregistrent leur activité spécifique. Ils proposent généralement de configurer un niveau d'enregistrement, selon l'importance des événements, afin de limiter la taille des rapports. S'il est possible d'analyser ces *logs* individuellement, la plupart des documents recommandent aujourd'hui [48, 46] de les collecter sur une machine spécialisée qui se charge de les analyser. De cette manière, on profite de la spécificité de chacun des enregistrements, tout en déchargeant chaque machine de la tâche d'analyse, qui est réalisée par une instance dédiée, pouvant donc utiliser plus de puissance et effectuer des

1. SysAdmin, Audit, Network, Security

analyses plus détaillées. Kakuru [36] recommande même d'ajouter un composant qui recopie toutes les données transmises sur le réseau pour y étudier les habitudes et le comportement des utilisateurs, afin de détecter les attaques venant de l'intérieur du réseau.

La quantité de données produites rend le stockage de plus en plus difficile. Dans leur revue [7], Broder et Mitzenmacher citent plusieurs applications qui utilisent les filtres de Bloom (voir la section 2.2.2) pour condenser les données à stocker et accélérer leur traitement. Ponec *et al.* [57] en reprennent quelques évolutions plus récentes.

Analyse précise des paquets

Plusieurs outils on été rapidement proposés pour analyser ces immenses bases de données d'actions effectuées sur le réseau, permettant de détecter plus efficacement et plus rapidement les éléments suspects. Kemmerer et Vigna [38] divisent ces outils en deux catégories : la détection d'anomalies d'une part, qui effectue une analyse du comportement des utilisateurs et relève les irrégularités, et la détection d'abus ou de mauvaises utilisations, qui compare le trafic à une liste de descriptions d'attaques connues (qu'il faut donc constamment tenir à jour) pour signaler les actions interdites. Les problèmes classiques sont les intrusions ou accès non-autorisés [69, 80], ou encore les transmissions de logiciels malveillants ou de données sensibles [78].

Toutes ces analyses sont effectuées après les incidents et visent à comprendre les moyens et les failles utilisés, pour finalement améliorer la sécurité du réseau. C'est donc une analyse indispensable, mais qui ne paraît pas adaptée à notre système car nous souhaitons bloquer des contenus lorsque l'utilisateur les charge, en temps réel, sans utiliser de blocage permanent d'un utilisateur ou d'une adresse IP. Ce système pourrait en revanche s'ajouter aux enregistrements cités plus haut, permettant de cibler les sources de contenu illicite pour ajouter si besoin des limitations plus globales *a posteriori*.

2.3.2 Analyse en temps réel

À la suite de ces systèmes *offline*, beaucoup d'applications ont été développées pour la détection de contenu connu. Un grand effort a été lancé dans la rapidité des traitements, et l'on a finalement pu trouver des programmes de filtrage de flux en temps réel destinés à détecter, voire à bloquer du contenu illicite ou suspect au moment de son passage sur le médium surveillé. Il peut s'agir de mesurer, d'analyser et d'optimiser le trafic réseau, application bien étudiée en 2009 dans la revue de littérature de Callado *et al.* [8], de prévenir les fuites de données personnelles ou de documents sensibles [78] ou encore détecter de potentielles intrusions dans un réseau, systèmes que l'on appelle "*Intrusion Detection System*

(IDS)" ou "*Intrusion Prevention System* (IPS)" [69, 80]. Ces systèmes essaient de reconnaître rapidement des signatures connues dans des paquets de données, sur un réseau par exemple. Il s'agit alors de tester le contenu de chaque paquet individuellement pour le comparer avec les modèles, méthode que nous présenterons plus loin.

De manière générale, il est toujours préférable d'agir préventivement contre les problèmes. Dans ce contexte, il est intéressant d'analyser en temps réel des flux de données pour y repérer très rapidement toute activité douteuse. Cette analyse entre en jeu à plusieurs niveaux, particulièrement pour la traque de contenus connus (illicites, interdits, d'accès limité, etc.) :

- accès à Internet : analyse du trafic entrant et sortant
- serveur d'entreprise : analyse des accès, des connexions et du contenu stocké
- mémoire des machines des utilisateurs : analyse des programmes en cours d'exécution et du contenu traité

Nous nous intéresserons dans cette partie aux différentes méthodes qui existent afin d'avoir une idée précise de l'architecture optimale pour un système de blocage.

Analyse de mémoire

Les systèmes d'analyse les plus connus sont les logiciels antivirus. Ils sont présents sur la majorités des postes utilisateurs mais aussi sur la plupart des serveurs de courriel ou autres types de stockage de contenu. Ces logiciels analysent généralement la mémoire vive en temps réel ou à l'accès, et parfois le contenu de la mémoire de stockage. Il existe plusieurs bases de données de signatures de logiciels malveillants, la plus connue étant ClamAV [12], qui regroupe plus d'un million de signatures (1 272 286 au 25 juillet 2012) et est publique. Il suffit donc d'analyser la mémoire vive de l'ordinateur lorsque les programmes y accèdent pour vérifier qu'aucune de ces signatures ne peut y être trouvée. Dans le cas contraire, on peut rapidement bloquer l'exécution du fichier mis en doute, voire retracer sa provenance pour éventuellement en bloquer la source.

Beaucoup de signatures sont des expressions régulières, dont nous avons parlé plus haut, ou des chaînes de caractères simples, qu'il suffit de rechercher dans les données à analyser. Cette base de données est donc très largement utilisée [20, 21], et les implémentations peuvent être facilement optimisées suivant le matériel visé (voir la section 2.3.3 : Matériel spécialisé). De son côté, Google [27] fournit une base de données (1 475 735 signatures au 25 juillet 2012) de signatures calculées sur des segments de contenu compromis, la documentation parle de *phising* et *malwares*. Ces signatures sont destinées à être calculées et comparées par les navigateurs lors du chargement des pages web. Il s'agit donc ici de limiter les risques pour l'utilisateur en dernier recours : les données sont examinées lorsqu'elles arrivent à l'écran.

Deep Packet Inspection

À plus haut niveau dans les réseaux informatiques, on met souvent en place des systèmes de détection d'intrusion afin de repérer des actions suspectes en provenance de l'extérieur de ces réseaux. Il s'agit le plus souvent d'usurpation d'identité ou de droits d'accès. La méthode utilisée dans la plupart des systèmes actuellement consiste à analyser chaque trame Ethernet pour en inspecter le contenu selon différentes règles. On parle alors de "*Deep Packet Inspection*". Cette inspection est souvent confiée aux *firewalls*, mais à mesure que les demandes deviennent plus nombreuses et plus précises, un système dédié peut s'avérer indispensable pour un traitement efficace.

Il s'agit ici de lire le contenu de chaque trame Ethernet et de le traiter selon différentes heuristiques pour finalement le déclarer sain ou à risque. De la même manière que pour les antivirus, cette méthode peut être efficacement liée à des bases de données de modèles d'attaques. Snort [62] est la base de données la plus utilisée dans les systèmes de détection d'intrusion (IDS). Il s'agit d'un répertoire mis à disposition du public, regroupant plus de 20 000 règles, composées majoritairement d'expressions régulières ainsi que de quelques chaînes de caractères simples. Les implémentations sont nombreuses [58, 80] car il s'agit d'un composant essentiel de la sécurité réseau.

Le point faible de ces méthodes est en général leur impact sur le réseau. En effet en travaillant à des débits de l'ordre de 10 Gb/s, il faut disposer d'algorithmes d'analyse très rapides pour analyser chaque paquet [9]. Dans le cas contraire, les paquets seront bloqués trop longtemps et l'utilisation du réseau en sera négativement impactée. Les approches matérielles disposent d'un avantage dans ce domaine car leurs latences sont relativement faibles. Nous étudierons plus loin les implémentations qui existent dans la littérature.

Notons que ce procédé est très controversé du fait de sa capacité à lire le contenu de toutes les communications Internet. Nous présenterons cet important débat dans la section 5.4. Néanmoins, le principe est intéressant car il consiste à analyser individuellement le contenu des paquets entrants et sortants. Nous pouvons exécuter l'algorithme de détection de fragments connus sur ces contenus pour détecter les communications interdites. En effet, lorsque l'on repère un fragment illicite dans un paquet, il suffit d'en lire les entêtes pour finalement signaler la source, la destination, ou tout autre paramètre utile.

2.3.3 Matériel spécialisé

Le premier cahier des charges que nous nous sommes fixé (voir section 2.2) demandait de travailler à l'échelle des paquets Internet, en pouvant repérer un grand nombre de documents différents, avec une bande passante de plusieurs dizaines de gigabits par seconde. Il est donc

intéressant de mesurer ces différents points sur les principaux algorithmes et implémentations disponibles dans la littérature. L'algorithme de *Max-Hashing* que nous allons utiliser correspond bien à ceux étudiés par Fowers *et al.*. Ils ont pu comparer les performances entre différents matériels parallèles, notamment FPGA et GPU, et concluent qu'ils ont chacun leur domaine d'efficacité en terme de quantité de données à traiter [22], les GPU étant plus rapides lorsque les fenêtres de calculs sont petites (jusqu'à 128 pixels) tandis que les performances du FPGA sont indépendantes de la taille des fenêtres, ce qui les rend intéressants pour des tailles plus importantes. Les performances des différentes implémentations que nous citerons sont extraites et regroupées dans le tableau 2.1.

Surveillance réseau

Dans le cadre d'une connexion réseau il est souvent primordial de garder une latence très faible. Ceci encourage l'utilisation de matériel dédié. Ainsi, dès 2002 on a vu apparaître des applications de *string matching* sur FPGA [10, 26, 30]. Ces implémentations peuvent faire des recherches de plusieurs milliers d'expressions régulières sur des paquets réseaux en gardant des latences inférieures à la milliseconde. Les bandes passantes peuvent atteindre le gigabit par seconde, mais les systèmes doivent être totalement embarquées car les bus PCI ne peuvent alors pas transmettre à cette vitesse. Les vitesses de connexion évoluant, des implémentations pour de plus hauts débits sont apparues [37].

Les performances du matériel sont principalement obtenues grâce au parallélisme des traitements puisque chaque test d'une expression régulière peut être lancé indépendamment

Tableau 2.1 – Résultats observés dans la littérature

Algorithme	Bande Passante	Type de Mesure	Latence	Règles
[73] GPU - Snort	2.3 Gb/s	Total	?	1000
[35] GPU - Snort	50 Gb/s	GPU	?	2448
[74] GPU - ClamAV	20 Gb/s	Total	?	60 000
[72] GPU - Expressions régulières	22 Gb/s	Total	?	20 000
[75] GPU - Snort	126 Gb/s	GPU	?	8719
[77] GPU - Snort	28 Gb/s	Total	?	8719
[44] GPU - Snort	143 Gb/s	GPU	?	8722
[79] FPGA - Snort	12 Gb/s	Total	?	256

Rappel : Snort *et* ClamAV *contiennent des chaînes de caractères et expressions régulières.*
Les latences ne sont malheureusement pas mesurées dans les articles relevés ici.

des autres. Si les FPGA et puces similaires apparaissent très indiquées pour ces applications, il semble intéressant de mentionner les processeurs graphiques, conçus pour effectuer des tâches en parallèle. De plus, leur accessibilité liée à la simplicité de programmation à la volée les rendent beaucoup plus attrayants que des puces dédiées. L'algorithme de *Max-Hashing* ayant déjà été implémenté sur FPGA, effectuer les tests sur une plateforme GPU présente aussi l'intérêt de pouvoir être comparé facilement.

Depuis 2006, des tests sont conduits sur les programmes de détection d'intrusion pour évaluer le gain de performance offert par le transfert de la recherche de signatures du CPU vers un GPU. On annonce alors des gains de 1.4x en 2006 [34], puis de 2x en 2008 [73], 9x en 2009 [67], jusqu'en 2011 lorsque Wang *et al.* annoncent [75] que la recherche d'expressions régulières traite les données avec une bande passante plus grande que celle autorisée par le bus PCI Express (PCIe) sur lequel est branché le GPU. Le traitement global, c'est-à-dire depuis la mémoire RAM du CPU jusqu'à obtention de la liste des correspondances, incluant ainsi le transfert vers la mémoire du GPU, accepte alors 25.6 Gb/s. Plus récemment, Liu *et al.* [44] ont implémenté leur algorithme (dérivé de l'algorithme d'Aho et Corasick [2]) sur GPU avec les règles de Snort et ont observé que celui-ci pouvait traiter jusqu'à 143 Gb/s sur une carte graphique NVidia GTX580, mais là encore ce traitement est ralenti par le bus PCIe à 15 Gb/s. Cette vitesse particulièrement faible est sans doute causée par un manque d'optimisation des transferts, nous aborderons cette question au chapitre 3.

Antivirus

Avec de tels débits de traitement, il devient intéressant d'utiliser les GPU comme co-processeurs dans la recherche de signatures connues. Si les articles précédents traitaient principalement de la détection d'intrusions dans un réseau, on peut de la même façon utiliser des bases de données de virus ou *malwares* connus pour créer un coprocesseur antivirus. Une autre méthode intéressante a été proposée. En considérant que le calcul d'expressions régulières est relativement complexe, il s'agit de rechercher des signatures fixes telles que des signatures pour créer un pré-filtrage des paquets, pour ensuite les soumettre à analyse plus approfondie en cas de nécessité. Erdogan et Cao testent cette approche en 2007 [20]. Ils utilisent la base virale de Clam-AV, dont nous avons parlé à la section 2.3.2, et parviennent à scanner les fichiers à l'accès avec une bande passante de 1.6 Gb/s. En 2009, Lin *et al.* déchargent le CPU des tâches de calcul et recherche des signatures vers un processeur spécialisé [43]. Celui-ci peut interrompre le CPU lorsqu'un danger potentiel est détecté. Les auteurs signalent très vite les latences et ralentissements impliqués par le transfert des données de la mémoire centrale à la mémoire embarquée. Leurs simulations présentent un taux de traitement théorique de 9 Gb/s en utilisant la recherche de chaînes de caractères de base.

En 2010, on revient finalement à la limitation des débits par les transferts de données lorsque Vasiliadis et Ioannidis parviennent à implémenter sur un GPU l'analyse de fichiers en temps réel, c'est-à-dire lorsque le fichier est utilisé, avec des performances de 20 Gb/s [74]. La vérification par expressions régulières, qui est lancée en complément des détections initiales par le GPU, est cependant laissée sur le CPU car elle n'est requise que dans de très rares cas. On constate aisément lorsque l'on mesure les performances totales d'un système qu'elles sont beaucoup moins importantes que la partie GPU seule, ce qui pointe clairement l'impact des transferts de données.

Évolutions rapides du matériel

Les GPU sont efficaces pour le traitement de paquets réseaux, l'indépendance entre ces derniers étant parfaitement adaptée aux immenses capacités de parallélisme de ces processeurs. Les expériences montrent qu'ils sont capables de traiter les paquets avec quelques milliers de règles à plus de 100 Gb/s sur les modèles hauts de gamme, même avec des expressions régulières, beaucoup plus complexes que des recherches de textes simples. Le bus PCIe dans sa version 2.0 16x peut faire transiter théoriquement 64 Gb/s en full duplex. Nous allons donc utiliser ce type de matériel pour surveiller une connexion Ethernet à 40 Gb/s.

De plus, la version suivante du bus PCIe (v3.0) double ces bandes passantes [56], ce qui peut laisser entrevoir le traitement à 100 Gb/s dans un futur proche et nous laisse donc penser que cette technologie présentera les performances adéquates. En outre, il s'agit d'une solution économique, car le ratio puissance/prix des GPU est avantageux, notamment par rapport aux CPU et FPGA.

La suite de ce mémoire s'intéressera ainsi à l'implémentation de l'algorithme de *max-hashing* sur GPU.

CHAPITRE 3

ARCHITECTURE PROPOSÉE

Comme nous l'avons annoncé au chapitre précédent, nous allons baser le système de filtrage sur l'algorithme de *Max-Hashing* pour analyser le contenu des paquets de données, tout en tirant partie de la puissance de calcul très importante et massivement parallèle des processeurs graphiques (GPU). Nous diviserons ce chapitre en quatre parties. Tout d'abord dans la section 3.1 nous expliquerons comment récupérer une copie du trafic afin de l'analyser, et nous détaillerons les possibilités pour le contrôler en bloquant des connexions spécifiques. Ceci nous fournira des éléments du cahier des charges pour le système d'analyse. Nous présenterons alors précisément l'architecture des GPU dans la section 3.2 afin de disposer de tous les éléments pour une implémentation optimale de l'algorithme de *Max-Hashing* que nous développerons à la section 3.3. Nous pourrons alors l'intégrer dans un système complet à la section 3.4. Les tests et évaluations de celui-ci seront reportés au chapitre 4.

3.1 Manipulations sur le trafic réseau

Nous souhaitons ici analyser le contenu de chaque paquet qui transite sur une connexion réseau. Pour ce faire, il convient bien entendu d'être nous-même connecté à ce réseau. Deux choix s'offrent à nous : soit placer un routeur qui se contenterait de recopier l'ensemble du trafic et de le transmettre à notre machine, soit placer directement notre machine sur la connexion en question (voir sur la figure 3.1). Les deux méthodes doivent absolument être totalement transparentes pour le réseau, c'est-à-dire qu'elles ne doivent pas demander de

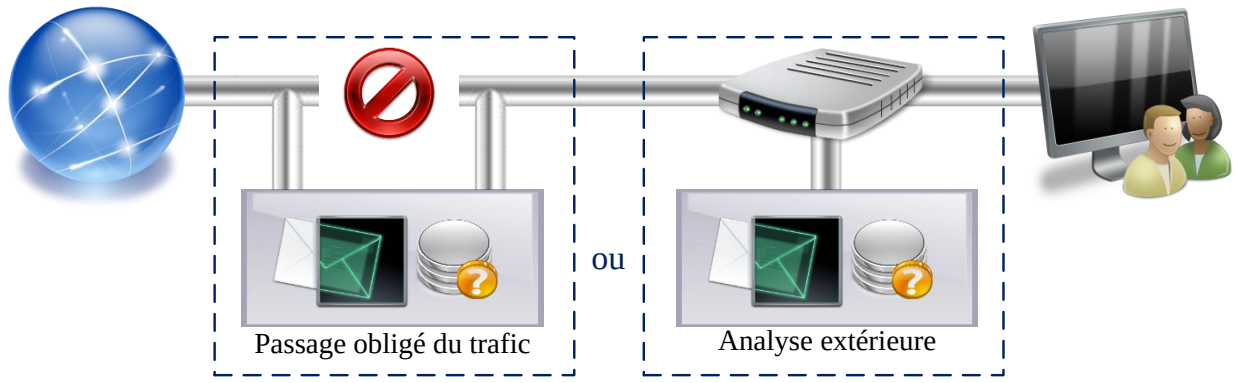


Figure 3.1 – Deux options pour surveiller une connexion

modification du reste des systèmes connectés lorsque l'on insère, active, désactive ou enlève le filtre.

L'avantage à utiliser un routeur ou du matériel spécialisé est de pouvoir s'appuyer sur le savoir-faire du fabriquant et sur ses garanties de fonctionnement et de fiabilité. Pour autant, n'ayant pas accès à du matériel avec de telles fonctionnalités, nous allons dans cette section présenter une solution entièrement logicielle. Il s'agit d'une solution de dépannage uniquement, nous le verrons dans l'analyse des performances au chapitre 4, mais elle nous offre un aperçu du fonctionnement final du système complet. De plus, on utilise exclusivement le système d'exploitation, ce qui simplifie beaucoup la configuration. Trois points vont être abordés : la capture du trafic, l'interconnexion de deux ports ethernet, et le filtrage de la connexion.

3.1.1 Écouter l'ensemble du trafic

Comme on l'a expliqué au paragraphe 2.1.1 (Connexion Ethernet), le contenu des données est, au plus bas niveau présenté ici, intégré dans une trame Ethernet. Celle-ci contient notamment les adresses MAC de l'expéditeur et du destinataire. Les machines connectées au réseau reçoivent l'ensemble du trafic et doivent comparer les adresses visées avec leur propre adresse pour n'enregistrer que les trames qui leur sont destinées. Ce système pourrait nous empêcher d'analyser les transferts n'impliquant pas notre machine (tous les transferts *a priori*). Heureusement, les concepteurs de processeurs réseau permettent de désactiver ce filtrage en passant en mode "*promiscuous*". La carte réseau considère alors que toutes les trames Ethernet la concernent et les transmet à l'utilisateur, système d'exploitation ou autre. Ce mode est donc utilisé pour des systèmes comme que le nôtre, nécessitant d'analyser tous les paquets, quelles que soient leur provenance ou leur destination.

Des logiciels peuvent ensuite faire l'interface entre la mémoire de la carte réseau et le programme d'analyse. Il s'agit ici de récupérer une copie des paquets reçus par la carte réseau. La chaîne de base fournit les paquets au système d'exploitation, qui se charge alors éventuellement de les filtrer et de les transmettre aux différentes applications qui en ont fait la demande. Il est aussi possible de passer outre le système d'exploitation mais ceci requiert des manipulations dans le logiciel de gestion de la carte réseau.

Le logiciel "*pcap*" [70], développé par TCPDump, offre une interface simple pour configurer la carte réseau et le système d'exploitation afin qu'ils transmettent une copie du trafic à l'application (voir figure 3.2). Il suffit de spécifier une fonction à exécuter lorsqu'un paquet est prêt à être traité. Cette méthode est très simple à mettre en place, mais elle implique de copier plusieurs fois le paquet, ce qui est sous-efficace : la carte réseau les copie dans la mémoire du système d'exploitation, qui lui-même les recopie dans des espaces mémoire

```
1 // Ouverture et configuration de la carte reseau
2 pcap_t *capture = pcap_open_live( nom_carte_reseau,
3                                   taille_buffer,
4                                   mode_promiscuous,
5                                   timeout,
6                                   buffer_message_erreur );
7
8 // Configuration d'une fonction de callback pour traiter les paquets
9 pcap_loop( capture, nombre_paquets, fonction_callback, parametres);
```

Figure 3.2 – Capture du réseau avec pcap

dédiés à chacune des applications. Aller récupérer les paquets directement sur la carte réseau pourrait devenir intéressant pour obtenir des débits importants et réduire les latences. Ces points seront étudiés au chapitre 4.

3.1.2 Pont réseau

Comme on l'a signalé, pour récupérer les paquets qui transitent sur le réseau, on peut choisir entre s'intercaler sur la connexion à surveiller ou utiliser un routeur dédié, qui se chargera de dupliquer le trafic et de nous en transmettre la copie. La première possibilité présente l'avantage de tout gérer dans le même et unique système. Il suffit alors de disposer d'une machine avec deux ports pour pouvoir s'intercaler n'importe où sur un réseau. On dispose de plus du contrôle total du trafic qui transite par cette machine, ce qui nous permettra aisément de mettre en place le filtrage (voir section 3.1.3 ci-après).

Pour nos tests, nous utiliserons un ordinateur fonctionnant sous Linux. Ce système d'exploitation offre une gestion très fine du réseau, avec notamment la possibilité de configurer simplement un pont (logiciel) entre deux ports réseau, quels que soient leurs protocoles et leurs formats respectifs. Avec cette méthode on peut donc aussi jouer le rôle de passerelle entre deux types de réseaux fonctionnant différemment. L'inconvénient est que le système d'exploitation doit gérer de façon logicielle les données transmises, ce qui peut demander une grande puissance lorsque l'on veut travailler avec des bandes passantes élevées. Il est possible que l'on puisse observer un ralentissement de trafic suivant la charge de la machine. De même, la gestion logicielle est beaucoup plus lente qu'un traitement matériel, et elle introduit donc une latence beaucoup plus importante. Néanmoins, nous utiliserons ce système pour sa simplicité de mise en place, en gardant cette limitation à l'esprit pour les tests du système complet.

```
ebtables -A FORWARD                    // Selection du trafic sortant
         -p IPv4                       // Protocole (en-tete Ethernet)
         --ip-source 11.12.13.14       // Adresse IP de l'emetteur
         --ip-destination 1.2.3.4      // Adresse IP du destinataire
         --ip-protocol TCP             // Protocole (en-tete IP)
         --ip-source-port 54321        // Port de l'emetteur
         --ip-destination-port 80      // Port du destinataire
         -j DROP                       // Action a appliquer
```

Figure 3.3 – Exemple d'utilisation d'ebtables

3.1.3 Filtrage du trafic

Linux dispose d'outils de filtrage réseau applicables aux différentes couches (Ethernet, IP, TCP/UDP), appelées "*IPTables*" [54]. Cet outil est très bien documenté [1] mais ne gère que le trafic qui atteint la machine (en tant qu'expéditeur ou destinataire). Les données qui passent sur le pont réseau ne sont pas vues. Il existe l'équivalent pour cette configuration : "*ebtables*" [53] (*Ethernet Bridge frame table administration*). Il dispose de moins d'options mais peut malgré tout filtrer les couches réseaux et protocoles les plus communs. Comme on l'a expliqué au paragraphe 2.1.2 (Protocole TCP/IP), une connexion est définie de façon unique par l'adresse et le port de l'émetteur et du destinataire (quatre paramètres). Ces options sont disponibles dans *ebtables* [2] et l'on pourra donc l'utiliser pour empêcher les paquets visés de traverser le pont réseau, comme illustré à la figure 3.3.

On peut donc facilement filtrer les transmissions. Dès lors que du contenu illicite est repéré dans un paquet, on configure les *ebtables* qui se chargeront de bloquer la suite de cette communication. Le destinataire d'un fichier interdit recevra donc le début de ce fichier, mais jamais la fin. Suivant la latence de l'analyse, on ne pourra pas filtrer des fichiers trop petits, qui auront le temps d'être transmis totalement avant que le blocage ne soit mis en place. Cette configuration permet par contre d'introduire une latence minimale dans le réseau, puisque l'on ne bloque pas les paquets pendant leur analyse. Celle-ci est réalisée à part. On bloquera au besoin la suite de la transmission.

3.1.4 Résumé

Finalement, grâce aux outils de Linux et sur un seul ordinateur, on peut disposer d'une architecture qui :

1. Reçoit les paquets (trames Ethernet) sur une carte réseau ;
2. Copie les paquets dans la mémoire du système d'exploitation ;

1. Voir la page de manuel en tapant `man iptables` dans une console Linux.
2. Voir la page de manuel en tapant `man ebtables` dans une console Linux.

3. Transmet les paquets à notre application ;
4. Filtre les paquets par protocole, adresses IP et ports selon des règles données ;
5. Renvoie les paquets par une autre carte réseau.

Il reste donc à concevoir l'application qui se chargera de lire les contenus des paquets, les analyser, et configurer le filtrage (*ebtables*) en fonction des résultats.

Notons le nombre important d'opérations effectuées avec les paquets. Il faudra s'assurer que la machine utilisée soit capable de fournir autant de traitements à haut débit. Il est aussi possible d'utiliser un routeur pour recopier le trafic du réseau. Des routeurs évoluées pourront en outre s'occuper du filtrage, déchargeant ainsi notre machine.

3.2 Processeur graphique

Les processeurs graphiques (ou GPU) sont de plus en plus utilisés pour des calculs généralistes. Il ont d'abord été détournés de leur usage original par les programmeurs, puis ont finalement été adaptés directement par les constructeurs. Ces puces sont généralement déportées sur une carte fille branchée sur un bus PCI Express, et sur laquelle elles disposent d'une mémoire dédiée. Le GPU a été conçu pour fournir une puissance de calcul très importante, ajustée pour être efficace dans des environnements graphiques à haute-définition. On cherche à exploiter ces ressources en dehors du traitement graphique pour des calculs plus généraux. On parle alors de "*General-Purpose computation on Graphics Processing Units* (GPGPU)". Aujourd'hui, les ordinateurs disposent de plus en plus de cartes graphiques intégrées dans tous les modèles. Les programmeurs sont donc poussés à les utiliser, déchargeant ainsi le processeur principal d'opérations pour lesquelles le GPU est plus adapté. La démocratisation des cartes graphiques dans les ordinateurs et la généralisation de leur utilisation motive finalement leur développement rapide, en puissance, fonctionnalités, et simplicité de programmation.

Note : le vocabulaire que nous allons utiliser est celui proposé par NVidia [55] et son architecture CUDA que nous allons présenter dans cette section.

3.2.1 Introduction

L'architecture d'un GPU est centrée sur le parallélisme entre plusieurs petites opérations appliquées sur un grand nombre de données : il s'agit d'une architecture "*Single Instruction Multiple Data* (SIMD)". On dispose en effet de la possibilité d'exécuter en parallèle plusieurs centaines de processus qui vont tous travailler avec la même instruction simultanément, mais sur des espaces mémoire différents. L'utilisation de GPU est donc intéressante dès que l'on

cherche à exécuter la même séquence d'opérations sur un grand nombre de données, tel que l'ensemble des pixels d'un écran par exemple. Ces processeurs sont donc très adaptés pour les calculs matriciels (addition de deux matrices en une instruction), ou, plus proche de notre travail, pour le traitement de contenus indépendants. Les GPU dont nous disposons sont ceux de la série 2050, produits par NVidia fin 2010. Ils sont annoncés comme bénéficiant d'une puissance de calcul de 515 Gflops en virgule flottante double précision. En comparaison, les processeurs (CPU) de la même génération pouvaient fournir jusqu'à 100 Gflops environ (Comme par exemple le processeur Intel Core i7-980XE [76]). Les processeurs graphiques sont donc très puissants, mais ils sont en contrepartie moins généralistes que les processeurs à usage général.

Le constructeur propose CUDA, une architecture optimisée pour la programmation généraliste, sur laquelle repose le langage CUDA C, souvent abrégé en CUDA. Ce langage est propriétaire, mais NVidia a fait partie du groupe à l'origine du langage OpenCL [39]. Ce dernier se veut commun à tous les périphériques de calcul parallèle, tant les GPU que les CPU multicœurs, ou à terme les FPGA. La plupart des processeurs de la marque sont compatibles avec CUDA C, et OpenCL, plus portable, mais qui permet moins d'optimisations. Le langage permet aux programmeurs d'appréhender la programmation sans l'aspect graphique qui compliquait l'apprentissage dans les autres langages disponibles comme Cg ou OpenGL. Il est donc possible de créer rapidement des logiciels spécialisés et programmés sous la forme de noyaux ("*kernels*") qui seront exécutés en parallèle sur les nombreuses unités de calcul du GPU.

3.2.2 Fonctionnement matériel

Une carte graphique basée sur CUDA est constituée du GPU, d'un lien PCIe pour le relier au reste de l'ordinateur, et d'une mémoire dédiée. Dans le processeur, la répartition des ressources suit un modèle unique, mais le détail varie d'une version à une autre. Les versions sont nommées "*Compute Capability*", allant actuellement de la version 1.0 à la version 3.0 pour les processeurs les plus récents. Les données concernant ces architectures sont reprises dans le tableau 3.1.

Parallélisme

Le processeur est constitué d'un grand nombre de cœurs de calcul, regroupés en unités appelées "Multiprocesseurs" (voir la figure 3.4). Les multiprocesseurs sont indépendants entre eux. Ils ont chacun leur file d'exécution. La synchronisation des instructions exécutées par les *threads* est limitée à cette échelle (pas à l'ensemble du GPU). Ainsi, deux multiproces-

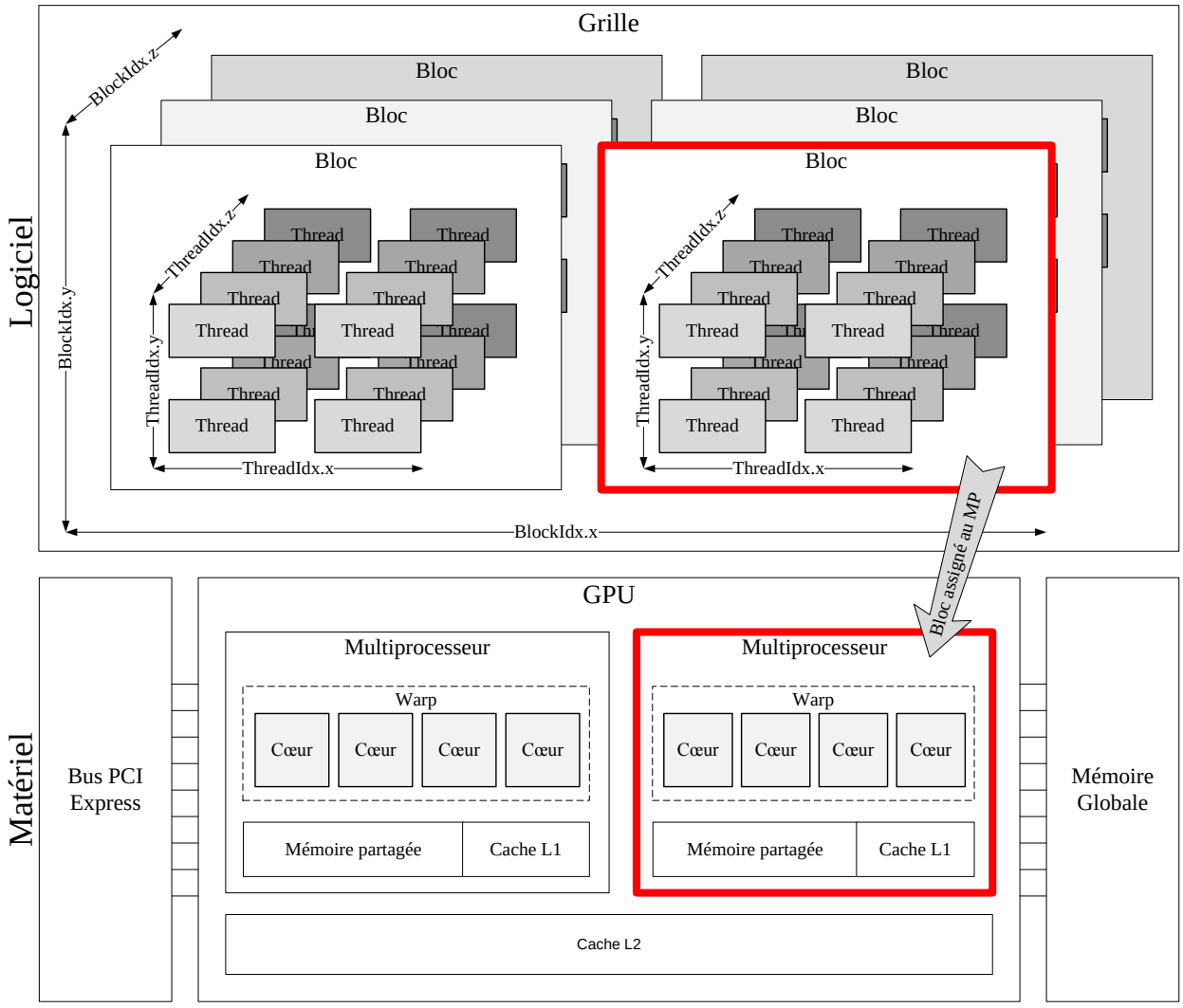


Figure 3.4 – Schéma de l'organisation d'un GPU

Tableau 3.1 – Configurations des différentes versions des processeurs NVidia

Version	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**	**2.1**	**3.0**
Threads par warp	32	32	32	32	32	32	32
Max. de warps par multiprocesseur	24	24	32	32	48	48	64
Mémoire partagée par MP (Ko)	16	16	16	16	48	48	48
Registres (Ko)	8	8	16	16	32	32	64
Répartition des registres	MP	MP	MP	MP	Warp	Warp	Warp
Max. de registres par thread	124	124	124	124	63	63	63
Max. de blocs par MP	8	8	8	8	8	8	16

Note : MP signifie Multiprocesseur

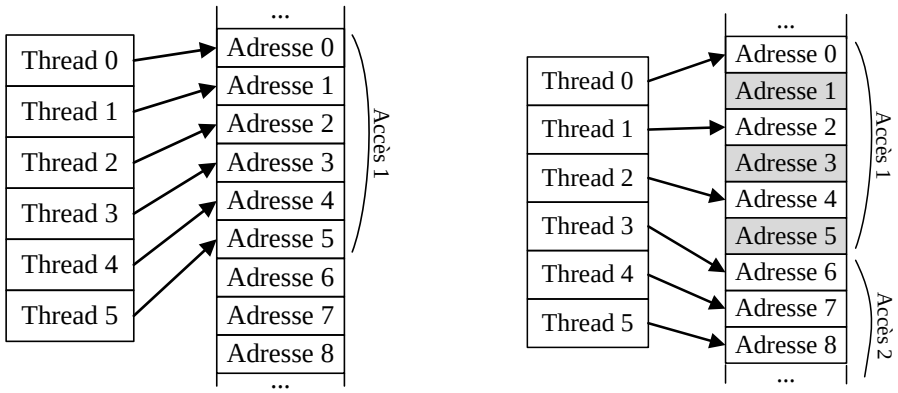
seurs peuvent exécuter deux fonctions différentes sur des données différentes si nécessaire. Les cœurs de calcul exécutent des *threads*, qui contiennent les instructions à réaliser, à l'instar des *threads* sur un CPU. L'ensemble des *threads* exécutés à un instant donné par un multiprocesseur, dont le nombre correspond au nombre de cœurs de calculs dans ce multiprocesseur, est appelé "*warp*". Le nombre de multiprocesseurs et de cœurs de calcul dépend de la version du processeur. la version 2.0 comprend par exemple 14 multiprocesseurs de 32 cœurs, et nous offre ainsi 448 cœurs parallèles. Il faut en connaître la configuration pour pouvoir l'utiliser à son maximum : il est par exemple peu efficace de lancer 64 *threads* sur un même multiprocesseur s'ils n'ont pas besoin de communiquer entre eux car seulement 32 seront exécutés ensemble, tandis que les 13 autres multiprocesseurs resteront inactifs.

Tous les cœurs d'un multiprocesseur exécutent la même instruction, il faut gérer les divergences de façon séquentielle. Cela signifie par exemple que l'exécution d'une instruction de branchement conditionnel sera effectuée en deux étapes, séparant les *threads* qui suivent ou non la condition du branchement. Il faut donc minimiser ces cas, sans quoi les divergences entre les *threads* peuvent vite conduire à une exécution totalement séquentielle, ne tirant plus aucun parti du parallélisme du processeur.

Mémoires

Les *threads* peuvent accéder à différents espaces mémoire, schématisés sur la figure 3.4. La plus importante en taille est la mémoire externe, souvent de la GDDR5, qui est accessible par l'ensemble des *threads*. Cette mémoire, appelée "mémoire globale" a une bande passante très importante, 144 Go/s dans le modèle C2050. Malgré deux niveaux de mémoire cache (L1 et L2) qui optimisent certains types d'accès, elle souffre d'une très grande latence (400 à 600 cycles selon la documentation de NVidia). C'est sur cette mémoire que sont copiées les données provenant de la mémoire centrale de l'ordinateur, et c'est sur cette mémoire que les fonctions devront effectuer leurs traitements. Les accès mémoire sont regroupés par *warp* (ou demi-*warp* sur certaines versions) et passent obligatoirement par les caches. Il est donc très important de s'assurer de l'alignement des données lues au sein d'un *warp* pour tirer profit de ces caches et limiter les accès inutiles. La figure 3.5 illustre l'effet d'une mauvaise organisation des lectures : la version non optimisée doit faire deux lectures et utilise uniquement 50% des valeurs lues tandis que la version optimisée n'a besoin que d'une lecture, qu'elle rentabilise pleinement. Selon les cartes proposées, on peut trouver de 1 à 6 Go pour la mémoire globale. La mémoire utilisée dans les cartes visant le calcul est souvent protégée contre les erreurs (ECC), contrairement à celle des cartes graphiques dédiées au jeu où l'on préfère la rapidité en tolérant des erreurs ponctuelles.

Il existe ensuite un espace mémoire, embarqué dans le GPU cette fois, lié à chaque



(a) Accès optimisés (b) Accès non optimisés

On cherche à aligner les accès à la mémoire pour tirer profit des caches.
Les adresses lues mais pas utilisées sont grisées.
On considère dans cette exemple une ligne de cache de 6 mots.

Figure 3.5 – Exemples de modèles d'accès à mémoire globale

multiprocesseur. Il s'agit de la "mémoire partagée". Tous les cœurs d'un multiprocesseur peuvent accéder à la mémoire partagée de celui-ci. Cette mémoire est beaucoup plus rapide et est divisée entre une partie accessible et une partie servant de cache de niveau 1 pour la mémoire globale. La quantité de mémoire disponible dépend de la version du processeur. Dans la version 2.0 il y a par exemple 48 Ko de mémoire par multiprocesseur, divisée en deux blocs de 32 Ko et 16 Ko à dédier à la mémoire partagée ou à la cache L1 (au choix du programmeur). La mémoire partagée est souvent utilisée lorsque les *threads* ont besoin d'accès croisés à la mémoire, par exemple lorsque tous les *threads* ont besoin de toutes les données. Il est beaucoup plus intéressant de commencer à copier toutes les données de la mémoire globale vers la mémoire partagée en optimisant les accès, puis de ne plus utiliser que la mémoire partagée, beaucoup plus rapide pour les accès aléatoires.

Finalement, les registres, en nombre plus limité, sont répartis entre les cœurs d'un multiprocesseur ou entre l'ensemble des *threads* exécutés sur un multiprocesseur dans les versions 1.x. Il est là encore capital de veiller à en restreindre l'utilisation, sans quoi le compilateur devra soit diminuer le nombre de *threads* exécutés en même temps, soit utiliser un espace de la mémoire globale, appelée alors "mémoire locale". Cette dernière est propre à chaque *thread* mais a les même limitations en temps d'accès et en alignement que la mémoire globale. Une particularité des registres est d'être un ensemble de 4×32 bits (le registre `a` est par exemple composé de `a.x`, `a.y`, `a.z` et `a.w`). Certaines opérations peuvent être effectuées très efficacement sur les quatre valeurs en même temps, il peut donc être utile de les connaître et de s'en servir, même si le compilateur gère cela très bien.

Transferts de données

Le GPU travaille avec les données situées dans la mémoire embarquée sur la carte graphique, qui lui est dédiée. Les données à traiter se trouvent généralement dans la mémoire centrale de l'ordinateur. Il faut donc d'abord les transférer avant que le GPU n'en ait besoin, et finalement rapatrier les résultats dans la mémoire centrale lorsque le CPU doit les exploiter. Calculer sur un GPU revient donc généralement à trois étapes : transmettre les données de la mémoire centrale vers la mémoire dédiée *via* le bus PCI Express, exécuter une fonction sur ces données, et récupérer les résultats en les copiant de la mémoire dédiée vers la mémoire centrale afin qu'ils soient exploités par le CPU. Le temps nécessaire aux transferts doit donc être pris en compte lorsque l'on veut effectuer une comparaison cohérente entre les temps de calculs locaux ou sur le GPU. Dans le tableau 2.1, qui résume les performances mesurées dans les articles cités au chapitre 2, on constate une différence importante de débit lorsque les auteurs prennent en compte les transferts ou non.

Notons que sur les versions récentes des cartes graphiques, les transferts de données entre la mémoire centrale et celle du GPU peuvent avoir lieu en même temps que les traitements et les accès à la mémoire du GPU. On peut alors créer une sorte de pipeline dans les applications, transférant les données d'un paquet alors que le précédent est en train d'être analysé. Les latences restent malgré tout identiques mais on peut utiliser le processeur et le bus à pleine charge. Le bus PCIe dans sa version 2 permet des transferts en full-duplex à 64 Gb/s avec 16 lignes, c'est la version utilisée actuellement sur les cartes graphiques. La version 3 double ces débits, et la version 4 les quadruple. On devrait donc pouvoir pour l'instant traiter des débits de 40 Gb/s. Il faudra attendre les cartes graphiques de la génération suivante pour travailler sur des connexions à 100 Gb/s. Une solution pourrait résider dans un système à plusieurs cartes graphiques, car le PCIe peut soutenir des transferts à pleine vitesse sur plusieurs bus en même temps. Il suffit alors de disposer d'une carte mère avec plusieurs bus et du contrôleur adéquat.

3.2.3 Programmation logicielle

On peut exécuter des fonctions logicielles sur le GPU directement. Ces fonctions, les *kernels*, utilisent et créent des données dans la mémoire globale. Un exemple est donné dans la figure 3.7 à la fin de cette section. Le *driver* des cartes graphiques gère tous les appels au GPU. Il le commande lorsqu'il s'agit d'initier les transferts mémoire, ou pour exécuter une fonction quelconque.

Les cœurs de calcul du GPU exécutent des fonctions appelées "*kernels*", écrites en CUDA C (ou autres langages tel que Cg ou OpenCL), qui travaillent à partir de la mémoire globale.

Ces fonctions ne retournent aucun résultat directement et ont accès à tous les étages de mémoire présentés au paragraphe précédent. Les données sources et les résultats sont placés dans la mémoire globale généralement sous la forme de tableaux de valeurs, et l'on exécute une instance du *kernel* sur chacune de ces valeurs pour traiter tout le tableau en parallèle. Des paramètres spéciaux peuvent néanmoins donner accès à la mémoire centrale de l'ordinateur pour les processeurs de version supérieure à 2.0 mais ceci bloque souvent le bus dans des petits accès lents et peu rentables.

Gestion du parallélisme

Lancer un calcul revient à créer un certain nombre d'instances des *kernels* appelées des *threads*. On crée en réalité une *grille* ("*grid*"), dans laquelle les *threads* sont répartis en *blocs* ("*blocks*") identiques. Lorsqu'on exécute une fonction sur le GPU, on ajoute donc deux arguments à ceux de la fonction : le nombre de blocs désiré, et le nombre de *threads* par bloc (voir figure 3.6). On crée donc une grille dont les paramètres sont transmis au *driver*. Celui-ci se charge alors de répartir les blocs entre les multiprocesseurs (voir la figure 3.4). Un bloc ne peut être divisé entre plusieurs multiprocesseurs, et un même multiprocesseur ne peut gérer que 8 blocs (16 pour la version 3.0, voir le tableau 3.1).

Comme on l'a vu plus haut, tous les *threads* d'un même bloc, qui seront donc regroupés dans un multiprocesseur, devront effectuer exactement les mêmes instructions pour être effectivement exécutés simultanément. Pour cela, les *kernels* peuvent faire appel à des variables qui indexent les *threads* dans les blocs et les blocs dans la grille, ce qui permet de précisément gérer le parallélisme. On dispose ainsi des variables suivantes, qui comprennent toutes trois composantes `x`, `y` et `z` :

- Configuration des blocs : `blockDim`
- Configuration de la grille : `gridDim`
- Index du *thread* dans le bloc : `threadIdx`
- Index du bloc dans la grille : `blockIdx`

```
1 // Configuration d'une grille de 2*4 blocs
2 dim3 blocs_grille( 2, 4, 1 );
3
4 // Configuration de blocs de 128*8*8 threads
5 dim3 threads_par_bloc( 128, 8, 8 );
6
7 // Lancement des threads dans cette configuration
8 ma_fonction <<< blocs_grille, threads_par_bloc >>> ( /* arguments */ );
```

Figure 3.6 – Lancement d'une fonction sur le GPU

```
1  // __global__ indique qu'il s'agit d'un kernel
2  __global__ void exemple(const int *a, int *b, int n)
3  {
4      // on identifie le thread sur le bloc
5      int thread_id = threadIdx.y * blockDim.x + threadIdx.x;
6
7      // premiere operation sur b
8      if(thread_id < n)   b[thread_id] = 0;
9
10     // on s'assure que tous les threads du bloc ont fini
11     // avant d'ecrire a nouveau dans b
12     __syncthreads();
13
14     // deuxieme operation sur b
15     if(2*thread_id < n) b[2*thread_id] = a[thread_id];
16 }
```

Figure 3.7 – Exemple de kernel en CUDA C

Ces indices peuvent notamment être utilisés pour calculer, avec les mêmes instructions, une adresse de données à traiter différente pour chaque *thread*. Dans l'exemple de la figure 3.7, on considère l'identifiant du *thread* au sein de son bloc uniquement pour clarifier le code.

Vu du CPU, les *threads* sont exécutés de façon asynchrone et les blocs sont traités dans un ordre quelconque, de même que les *threads* dans un bloc. Des moyens de synchronisation peuvent être utilisés à l'intérieur d'un bloc, principalement pour spécifier des barrières que tous les *threads* doivent atteindre avant de continuer l'exécution (voir figure 3.7). L'unique option pour synchroniser l'exécution de l'ensemble des *threads* de la grille est de créer plusieurs fonctions, qui seront appelées et exécutées les unes après les autres.

Le parallélisme peut aussi être géré à l'échelle de "*streams*". Il s'agit de piles d'exécutions incluant les transferts asynchrones, les lancements de fonctions et les opérations sur la mémoire. Ceci permet d'imposer l'ordre d'exécution d'opérations diverses. On peut créer plusieurs *streams* qui s'exécuteront en parallèle et indépendamment. Ceci permet de créer très facilement des opérations pipelinées, notamment transférer des données pendant que d'autres sont traitées, afin de masquer les temps de transferts.

Compilation et programmation

L'interface de programmation est très simple à utiliser, puisque le compilateur `nvcc` fourni peut gérer à la fois le code CPU (en appelant le compilateur dédié sur ces parties de code) et le code GPU (qu'il traite lui-même). Deux étapes de séparation ont lieu. Les fichiers sont tout d'abord répartis selon leur extension, `.cu` vers `nvcc` et `.c`, `.cpp` ou équivalents vers la chaîne de compilation pour le CPU. Ensuite, les fonctions destinées au CPU dans

des fichiers `.cu` sont elles-aussi transmises à ces mêmes outils. Les opérations de compilation ciblées pour chacun des processeurs sont donc totalement transparentes.

Le langage proposé par NVidia permet d'accéder à toutes les fonctionnalités de leurs processeurs, et donc de gérer très précisément toutes les optimisations. Ce dernier point constitue le principal avantage face à OpenCL. En effet ce langage est compatible avec d'autres cartes graphiques et des processeurs à usage général. Cette compatibilité permet de créer des applications portables sur des plateformes très diverses, mais masque les fonctions qui sont spécifiques à l'un ou l'autre des processeurs et autorise donc moins d'optimisations.

3.2.4 Conclusion

À la lumière des éléments précédents, nous pouvons aisément prévoir les défis que nous aurons à relever en programmant le processeur graphique. Les principaux points sont les suivants :

- Maximiser le parallélisme : pour un maximum d'efficience, il faut lancer assez de *threads*, répartis en blocs, pour que l'ensemble du GPU soit occupé en permanence.
- Minimiser les divergences : tous les *threads* doivent exécuter exactement les mêmes instructions, au moins au sein des blocs.
- Masquer les latences de la mémoire : en lançant beaucoup de *threads*, on peut maximiser le nombre d'opérations qui garderont le processeur occupé pendant que certains *threads* attendront le résultat d'un accès mémoire.
- Aligner les accès mémoire : ces accès présentant une latence importante, il sera crucial de veiller à l'alignement des requêtes notamment et à minimiser les lectures aléatoires, afin de réduire le nombre d'accès nécessaires.
- Prendre en compte les transferts : les mesures devront inclure les temps de transfert entre la mémoire centrale et la mémoire dédiée.
- Surveiller l'occupation en registres : s'il n'y a plus de registre disponible, le compilateur déportera le stockage des intermédiaires de calcul dans la mémoire DRAM. Il faut donc surveiller ces ressources, et il peut être utile de scinder le *kernel* en deux pour optimiser l'exécution globale.
- Optimiser les ressources : plus généralement, il faut veiller à utiliser les ressources disponibles, en particulier les unités de calcul, pour optimiser l'exécution. CUDA C donne accès simplement à la plupart des ressources spécifiques (celles qui ne sont pas utilisable directement peuvent l'être via l'assembleur). La documentation est très détaillée sur ce sujet [55].

On dispose de plusieurs méthodes pour analyser les *kernels*. NVidia fournit un outil de profilage qui donne de bonnes indications sur les modes d'accès à la mémoire, sur les di-

vergences, sur les optimisations des transferts, etc. et qui s'avère particulièrement utile. Le compilateur peut aussi fournir un équivalent du programme en pseudo-assembleur, permettant d'analyser la compilation et les espaces mémoire utilisés. Les principales optimisations seront effectuées au moment du lancement pour s'adapter au matériel disponible.

La suite de ce chapitre détaillera les adaptations de l'algorithme de max-hashing pour suivre ces différents points.

3.3 Adaptation de l'algorithme de max-hashing

Maintenant que les bases du fonctionnement et de la programmation des GPU sont posées, nous allons nous concentrer dans cette section sur le développement d'une version de l'algorithme de *Max-Hashing* optimisée pour ces processeurs. Comme mentionné au chapitre 2, l'algorithme a été implémenté en logiciel. Une adaptation sur FPGA a aussi été développée, pouvant traiter 10 Gb/s. Ainsi, Nous commencerons par citer les méthodes utilisées dans ces implémentations, notamment pour le calcul des signatures et la création de la base de données de signatures. Nous les adapterons ensuite aux ressources et spécificités des GPU.

L'algorithme de *Max-Hashing* se décompose en deux temps : le référencement de contenus à détecter d'une part, et l'analyse en temps réel d'autre part. Ces deux phases demandent un ensemble de données à analyser, calculent les signatures de cet ensemble, puis vont respectivement enregistrer ou rechercher les signatures calculées dans une base de données de référence.

Notons que le code de la partie analyse n'est pas repris ici pour des raisons de lisibilité. Il est toutefois disponible en annexe.

3.3.1 Calcul des signatures

Le calcul des signatures est la base de l'algorithme. On souhaite produire une signature qui représentera les données d'entrée. Le principe du calcul consiste à déplacer une fenêtre de

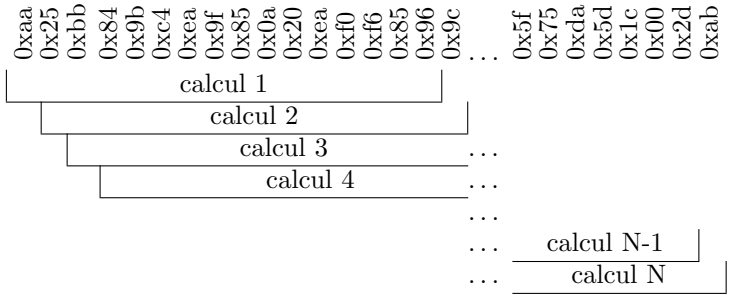


Figure 3.8 – Les calculs de signatures dans le max-hashing

sélection de taille fixée sur les données à traiter. On calcule alors la signature de cette fenêtre à chacune des étapes, comme illustré à la figure 3.8, grâce à une fonction préalablement fixée. Ce calcul représente une grande partie des traitements et il est donc primordial de l'optimiser. Il ne reste en effet après ce calcul qu'à chercher la signature de valeur maximale pour obtenir la signature finale, ce qui demande peu de traitement et peut même être réalisé au fur et à mesure du déplacement de la fenêtre.

Choix des tailles des fenêtres et des signatures

La signature alors calculée va jouer le rôle de représentant des données sources. On souhaite donc qu'elle soit facilement détectable, et aussi hautement représentative de ces données.

Le choix des tailles des données doit être étudié, tant pour les fenêtres de calcul que pour les signatures calculées sur ces fenêtres. En effet, il influera directement sur la qualité des résultats : une signature trop petite pourrait entraîner beaucoup de fausses alertes à cause du manque d'originalité de la signature référencée, tandis qu'une fenêtre trop grande réduirait la capacité à détecter de petits fragments du fichier... Dans les implémentations de test et de démonstration du *max-hashing*, une signature de 64 bits était calculée à partir d'une fenêtre de 128 bits. Les tests impliquant plusieurs dizaines de milliers de fichiers montraient un taux de faux positifs ou de faux négatifs proche de zéro.

En outre, il est possible de choisir uniquement les blocs hautement représentatifs du fichier lors de son référencement pour maximiser l'entropie de la signature. Les zones compressées des documents, notamment dans les images, sont par exemple de bonnes candidates. De même, une analyse statistique des signatures calculées sur plusieurs fichiers peut facilement indiquer quelles parties de ces fichiers sont redondantes, pour ensuite les exclure des zones d'intérêt.

Nous utiliserons dans un premier temps les mêmes formats de signatures, leur choix ayant été validé sur des images JPEG lors de l'élaboration de l'algorithme. Ceci permettra en outre de garder la compatibilité entre les différents systèmes, leur offrant une interopérabilité qui pourra être très pratique lors de tests. Nous validerons ces choix lors de l'analyse finale du système.

Rolling-hash

Dans la version existante, les signatures générées lors du déplacement de la fenêtre ne sont pas calculées indépendamment les unes des autres : la signature d'une fenêtre est déduite à partir de celle de la fenêtre précédente. On utilise le principe du *rolling hash* : on ajoute

à chaque étape la contribution à la signature de l'octet entrant, et on retire celle de l'octet sortant, afin que la signature obtenue ne dépende bien que de la fenêtre source et non des données avoisinantes. Une fenêtre de 128 bits est déplacée sur les données $(a_n)_{0 \leq n < N}$, créant les signatures correspondantes $(h_n)_{0 \leq n < N-15}$ selon le schéma suivant :

$$\ldots\ a_{n-1}\ \boxed{a_n\ a_{n+1}\ a_{n+2}\ \ldots\quad a_{n+14}\ a_{n+15}}\ a_{n+16}\ \ldots$$
$$\Downarrow$$
$$h_n$$

Lors du passage de h_n à h_{n+1}, le nouvel octet a_{n+16} entre dans la fenêtre de calcul alors que l'ancien octet a_n en sort. L'empreinte correspondante est modifiée en conséquence, en supprimant la contribution de a_n puis en ajoutant celle de a_{n+16}.

La prise en compte de chaque octet est réalisée par le biais d'une fonction f, qui représenterait alors ce que nous avons qualifié plus haut de "contribution" ou "influence" d'un octet. On utilise en pratique un paramètre p de 64 bits tel que pour tout octet x :

$$\mathrm{f}\,(x) = (x \times p)\,\%\left(2^{64} - 1\right) \tag{3.1}$$

Le modulo $2^{64} - 1$ permet de limiter le calcul à 64 bits. On peut en effet utiliser le fait que $2^{64}\%\,(2^{64} - 1) = 1$, et ainsi calculer séparément les 64 bits de poids fort et les 64 bits de poids faible pour enfin additionner ces résultats, ce qui reste simple à implémenter matériellement.

On effectue en plus une rotation sur 8 bits du *hash* à chaque itération. Ainsi, la valeur de $\mathrm{f}\,(a_{n+16})$ ajoutée à l'entrée de a_{n+16} dans la fenêtre de calcul peut être directement soustraite seize étapes plus tard car elle est revenu à son emplacement d'origine. On obtient donc finalement la relation de récurrence, où l'opération "$x \lll n$" représente la rotation de x sur n bits :

$$h_{i+1} = (h_i \lll 8) - \mathrm{f}\,(a_n) + \mathrm{f}\,(a_{n+16}) \tag{3.2}$$

Une fois l'ensemble du bloc de données parcouru, il ne reste qu'à calculer le maximum de toutes les valeurs des signatures pour obtenir celle qui représentera le bloc en question :

$$h = \max_i\,(h_i) \tag{3.3}$$

Adaptation au calcul sur 32 bits

Le GPU ne dispose que d'unités de calcul, en particulier les multiplieurs, pour des entiers de 32 bits. Il est plus efficace de diviser les mots de 64 bits en deux mots de 32 bits et de calculer deux signatures de 32 bits plutôt qu'une seule signature de 64 bits. De même, on utilise deux paramètres de 32 bits à la place d'un unique paramètre de 64 bits.

On effectue pour chaque signature, avec les fonctions $\mathrm{f_{hi}}$ et $\mathrm{f_{lo}}$ les multiplications par chacun des deux paramètres de 32 bits :

$$\mathrm{f_{hi}}\left(x\right) = x \times p_{hi} \quad ; \quad \mathrm{f_{lo}}\left(x\right) = x \times p_{lo} \tag{3.4}$$

On travaille avec la fonction `xor`, qui est involutive [1] et donc avec la récurrence suivante :

$$h_{i+1}^{hi} = \left(\left(h_i^{hi} \ll 4\right) + \left(h_i^{lo} \gg 28\right)\right) \oplus \mathrm{f_{hi}}\left(a_n\right) \oplus \mathrm{f_{hi}}\left(a_{n+16}\right) \tag{3.5}$$

$$h_{i+1}^{lo} = \left(\left(h_i^{lo} \ll 4\right) + \left(h_i^{hi} \gg 28\right)\right) \oplus \mathrm{f_{lo}}\left(a_n\right) \oplus \mathrm{f_{lo}}\left(a_{n+16}\right) \tag{3.6}$$

On utilise ainsi uniquement des multiplications, décalages et ou-exclusifs sur 32 bits. On utilise ensuite l'opérateur maximum intégré sur le GPU, qui fonctionne en 64 bits. Pour réduire le nombre d'instructions, on place donc les deux mots côte-à-côte pour former la signature finale, sur 64 bits, utilisée dans la comparaison et sauvegardée pour utilisation ultérieure :

$$h = \max_i \left(h_i^{hi} \ll 32 \mid h_i^{lo}\right) \tag{3.7}$$

On arrive donc à l'algorithme de la figure 3.9. On raffine en plus cette méthode pour calculer quatre signatures, en séparant les maxima selon les deux premiers bits de leur valeur. On obtient alors quatre maxima, placés à différents endroits dans le bloc de données.

Parallélisme au niveau des paquets

On notera que si cette méthode de calcul est très simple et semble donc pouvoir être efficace sur le GPU, elle n'en utilise plus les capacités de parallélisme, puisque les signatures ne sont plus indépendantes et ne peuvent donc pas être calculées en même temps. Ce parallélisme pourra tout de même être mis en place au niveau des blocs de données, puisque chaque bloc sera cette fois indépendant, et pourra donc être traité en parallèle d'autres blocs. Ceci est intéressant au niveau de la réception des paquets réseau : il suffit de stocker quelques paquets puis de les envoyer au GPU pour les traiter en parallèle, chaque *thread* pouvant être dédié à un paquet (voir figure 3.10(b)).

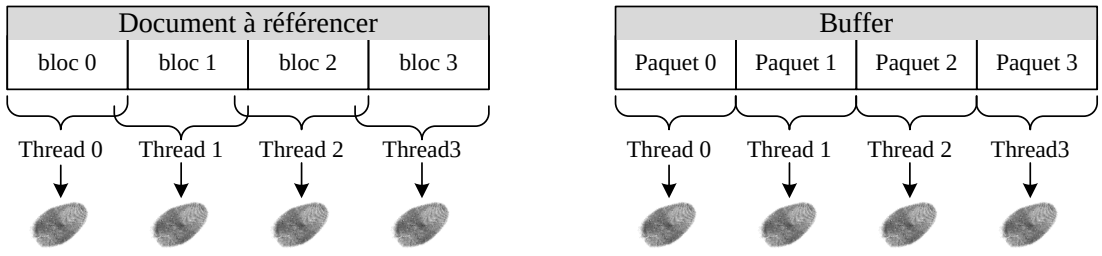
Nous implémenterons aussi la possibilité de ne pas traiter les documents complets, mais de pouvoir calculer une signature sur des blocs de taille fixe (voir figure 3.10(a)), ce qui permettra par exemple de contrôler précisément le nombre de signatures créées lors du référencement d'un fichier. De même, nous pourrons continuer dans l'idée de ne créer des signatures qu'à partir de blocs d'intérêt dans les documents, et non depuis l'ensemble du

1. $(a \oplus x) \oplus x = a$

Données: les données sources *source*, de taille D, la taille de fenêtre F
Résultat: la signature *sig*, la position *pos* dans la source

$sigTemp \leftarrow 0$
$sig \leftarrow 0$
$pos \leftarrow (-1)$
$h_{hi} \leftarrow 0$
$h_{lo} \leftarrow 0$
pour i **allant de** 0 **à** $D-1$ **faire**
  si $i < F$ **alors**
    $c_{out} \leftarrow 0$
  sinon
    $c_{out} \leftarrow source[i-F]$
  finsi
  $h_{hi} \leftarrow ((h_{hi} \ll 4) + (h_{lo} \gg 28)) \oplus \mathrm{f_{hi}}(c_{out}) \oplus \mathrm{f_{hi}}(c_{in})$
  $h_{lo} \leftarrow ((h_{lo} \ll 4) + (h_{hi} \gg 28)) \oplus \mathrm{f_{lo}}(c_{out}) \oplus \mathrm{f_{lo}}(c_{in})$
  $sigTemp \leftarrow h_i^{hi} \ll 32 \mid h_i^{lo}$
  si $sig < sigTemp$ **et** $i \geq F$ **alors**
    $sig \leftarrow sigTemp$
    $pos \leftarrow i$
  finsi
fin

Figure 3.9 – Principe du calcul des signatures



(a) Au référencement (b) Sur les paquets IP

Chaque *thread* calcule la signature d'un sous-bloc, avec un recouvrement de la taille de la fenêtre de calcul pour ne pas perdre de signature (lors du référencement seulement).

Figure 3.10 – Calcul des signatures

document.

Optimisation des accès à la mémoire

L'implémentation du calcul des signatures demande la plus grande vigilance car elle est difficile à optimiser, du fait notamment des accès mémoire nombreux et irréguliers ainsi que des effets de bord. Les *threads* lisent en effet les données de chacun des paquets IP qui leur sont attribués au fur et à mesure du calcul de la signature. On accède donc à la mémoire globale, qui contient ces paquets, sans aucun alignement des adresses lues puisque chaque *thread* demande un octet spécifique dans son paquet IP. Dès que les paquets sont plus grands que 128 octets, ce qui est très souvent le cas sur les paquets utiles, il faut effectuer plusieurs lectures. On gaspille donc beaucoup d'accès mémoire de la longueur d'une ligne de cache pour utiliser uniquement un octet.

Remarque : l'évolution du code en fonction des méthodes d'accès présentées dans ce paragraphe peut être suivie sur la figure 3.11.

L'implémentation directe consiste à lire chaque octet nécessaire directement dans la mémoire globale, tel qu'illustré sur la figure 3.11(1). On obtient des accès inefficaces car les emplacements lus simultanément sont trop espacés. On ne lit en effet qu'un octet par ligne de cache de 128 octets. On ne peut pas désactiver l'ensemble des caches, mais une première optimisation très simple consiste à désactiver la cache L1 qui impose les lectures de 128 octets, en n'utilisant alors plus que la cache L2. Celle-ci, tout en étant légèrement plus lente, présente l'avantage de n'utiliser que des lignes de 32 octets. On utilise ainsi légèrement plus de valeurs lues.

Les octets lus depuis la mémoire globale sont utilisés à leur entrée dans la fenêtre de calcul, et lus à nouveau lors de leur sortie. Une seconde optimisation consiste donc à enregistrer ces valeurs dans des registres lors de la première lecture pour ne plus avoir besoin d'appels à la mémoire globale. Un tableau de 16 octets peut ainsi être utilisé (voir figure 3.11(2)). On effectue deux fois moins d'accès mais on ne lit toujours uniquement un octet à la fois, n'utilisant donc qu'un octet sur les 32 lus dans la mémoire à chaque accès, soit un rendement de 3.125%. Pour améliorer ces résultats, il faudrait pouvoir mieux rentabiliser les accès de la cache à la mémoire, c'est-à-dire utiliser toutes les valeurs lues. L'instruction de lecture la plus grande disponible est de 16 octets. Il faut alors stocker les données dans des registres en attendant leur utilisation. On utilise alors 50% des accès réels à la DRAM en lecture. Notons qu'effectuer deux accès successifs pour utiliser 100% des valeurs lues demande de stocker trop de valeurs et les multiprocesseurs ne disposent plus assez de registres. Il nous paraît que la méthode de lecture actuelle est optimale. Le code résultant est traduit sur la figure 3.11(3), et utilise le type `ulonglong2` pour décrire un mot de 16 octets.

Toutefois, il reste la sauvegarde des 16 octets de la fenêtre courante introduite précédemment. Celle-ci étant actuellement un tableau de 16 octets, elle est stockée dans la mémoire locale car les registres ne peuvent être adressés dynamiquement. En gérant l'indexation directement par des décalages dans un vecteur, on permet au compilateur de placer la variable dans les registres. Les accès à la mémoire locale sont généralement très optimisés mais il n'en restent pas moins lents comparés aux accès à un registre.

1. Lecture d'octets avec `data` de type `char*` dans la mémoire globale

```
char c_in  = data[i];
char c_out = data[i-16];
mettre_a_jour_hash(c_in, c_out);
```

2. Sauvegarde des octets lus dans `last`, tableau de 16 octets dans la mémoire locale

```
char c_in  = data_in[i];
char c_out = last[i%16];
mettre_a_jour_hash(c_in, c_out);
last[i%16] = c_in;
```

3. Lecture de la plus grande quantité de données possible, soit 16 octets, avec `data16` de type `ulonglong2*` dans la mémoire globale

```
// Lecture uniquement toutes les 16 iterations :
ulonglong2 data_in = data16[i/16];
// Puis a chaque iteration :
char c_in  = (char)( data_in >> ( 8 * (i%16) ) );
char c_out = last[i%16];
mettre_a_jour_hash(c_in, c_out);
last[i%16] = c_in;
```

4. Sauvegarde des octets lus dans `last16`, de type `ulonglong2` (16 octets) stocké dans les registres

```
// Lecture uniquement toutes les 16 iterations :
ulonglong2 data_in = data16[i/16];
// Puis a chaque iteration :
char c_in  = (char)( data_in >> ( 8 * (i%16) ) );
char c_out = (char)( last16  >> ( 8 * (i%16) ) );
mettre_a_jour_hash(c_in, c_out);
// Sauvegarde a la fin des 16 iterations :
last16 = data_in;
```

Note : i représente la position dans les données sources, on utilise une fenêtre de calcul de 128 bits.

Figure 3.11 – Évolution des accès mémoire

Réduction des divergences

Le second problème du calcul des signatures est la divergence. À chaque nouvelle signature calculée, on la compare avec le maximum courant, que l'on remplace en cas de besoin. On peut utiliser l'instruction `max` disponible sur le GPU pour éviter simplement ce branchement. La figure 3.12(a) illustre la modification : au lieu de mettre à jour la valeur de la signature uniquement si la nouvelle valeur est plus importante que le maximum sauvegardé, on applique la fonction `max` à chaque fois, supprimant ainsi tout test. Les divergences apparaissent à nouveau lorsque l'on choisit d'enregistrer la position de la valeur maximale dans le paquet pour retrouver les données correspondantes. Il faut en effet enregistrer la nouvelle position uniquement si la signature est modifiée, ce qui nécessite un test et introduit donc un branchement.

Le calcul des quatre maxima peut aussi grever les performances en créant des divergences puisqu'il demande de sélectionner la valeur à comparer ou à remplacer avant d'effectuer le calcul en lui-même. Il y a alors quatre signatures et quatre positions à sauvegarder, dans l'un ou l'autre des espaces mémoire. Deux choix doivent être fait : stocker les signatures

```
1 // Une divergence directe :
2 if( sig < sigTemps ) sig = sigTemp;
3
4 // Avec un simple max, pas de divergence :
5 sig = max( sig , sigTemp );
6
7 // En enregistrant en plus la position , reintroduction de la divergence :
8 sig = max( sig , sigTemp );
9 if( sig == sigTemps ) pos = i;
```

(a) En sauvegardant la position

```
1 // Stockage des max dans un tableau , pas de divergence , mais memoire locale
2 int id = sigTemp >> 30;
3 sig[id] = max( sig[id], sigTemp );
4
5 // Stockage dans des registres , acces rapides mais divergences
6 int id = sigTemp >> 30;
7 switch( id ) {
8     case 0:
9         sig_0 = max( sig_0 , sigTemp );
10        break;
11 /* ... */
```

(b) En calculant plusieurs signatures

Figure 3.12 – Introduction de divergences

dans un tableau, facilement adressable sans branchement, mais devant être enregistré dans la mémoire locale, ou stocker dans quatre registres. Ceci impose de sélectionner dans quel registre a lieu la mise à jour. Nous étudierons l'impact de ces possibilités dans le chapitre 4. L'exemple présenté à la figure 3.12(b) illustre la traduction de ces choix en code ; on y utilise un `switch` pour la sélection du registre à mettre à jour, ensemble de tests qui crée une divergence puisque chaque processus se basera sur une valeur différente et empruntera donc un chemin différent.

3.3.2 Base de données

La base de données doit pouvoir stocker toutes les signatures, sans limite de taille. L'accès et la recherche d'éléments doivent être rapides et surtout sans perte de performance lorsque le taux de remplissage augmente. Le principe est simple : on référence les documents par des signatures qui sont alors stockées dans la base de données. Au moment d'analyser un paquet IP, on calcule la signature de son contenu, et on la compare avec celles qui sont stockées. En cas de correspondance, on peut considérer avec une forte probabilité que le contenu analysé provient d'un document référencé. On initiera alors les actions nécessaires.

L'implémentation la plus directe consiste à créer une *hash table* avec une fonction d'association très simple. On utilise les p bits de poids faible du *hash* pour indiquer le numéro de la ligne dans laquelle il sera stocké. La base de données prend alors la forme d'un tableau de valeurs de c colonnes par 2^p lignes, tel qu'illustré à la figure 3.13. On peut alors enregistrer c signatures avec les mêmes p bits de poids faible. Il est toutefois possible d'inclure un mécanisme pour gérer le débordement si une ligne est remplie, en ajoutant une colonne (un entier sur 8 bits par exemple) à la base de données, qui indiquera le nombre de lignes de débordement.

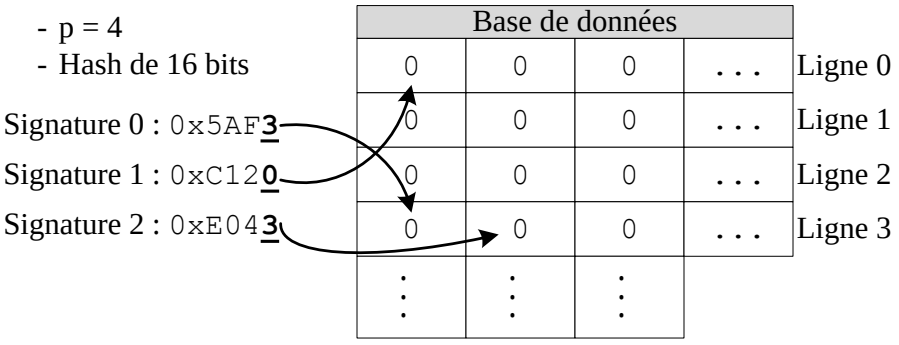


Figure 3.13 – Base de données de signatures

Une telle base de données utilise $8 \times c \times 2^p$ octets de mémoire (ou $(8 \times c + 1) \times 2^p$ octets avec le débordement), qui seront stockés dans la mémoire embarquée de la carte graphique pour un accès rapide par le GPU. Les deux paramètres c et p seront spécifiés suivant les performances et la taille nécessaire/disponible. Notons qu'il est possible de réduire l'espace de stockage puisque les p bits de poids faible n'ont pas nécessairement besoin d'être enregistrés si l'on peut les retrouver grâce à l'index de la ligne. On contrôle donc facilement la taille de cet espace de stockage et on sera en mesure de l'adapter aux différentes configurations que l'on cherche à réaliser.

Ajout d'une signature

Ajouter une signature à la base de données consiste à d'abord trouver la ligne dont le numéro correspond au p bits de poids faible de la signature, puis à lire les c colonne pour trouver la première "case" vide (c'est-à-dire de valeur nulle). On copie alors la valeur à cette adresse mémoire. Le principe est illustré sur la figure 3.13 : les deux premières signatures sont enregistrées au début des lignes correspondantes, tandis que la signature 2 doit être enregistrée dans la seconde colonne puisque la signature 0 se trouve déjà dans la première. On peut donc créer une fonction simple, qui lit en entrée une liste de signatures, et fournit en sortie le rapport d'exécution (par exemple la confirmation que la signature a été bien enregistrée). Il n'est pas réellement nécessaire d'optimiser la vitesse d'exécution de cette fonction puisqu'elle ne sera utilisée que lors du référencement d'un nouveau fichier, c'est-à-dire très ponctuellement.

Recherche d'une signature

La fonction de recherche est un point primordial de l'application. En effet, elle est autant appelée que le calcul des signatures pour vérifier la présence ou l'absence de chacune d'entre elles. Pour atteindre une signature, il faut tout d'abord trouver la ligne à partir des p bits de poids faible de la signature, puis comparer cette dernière avec les c valeurs enregistrées (potentiellement vides) de la ligne visée. L'avantage du GPU est ici la possibilité de lire toute la ligne en une instruction. Les accès mémoire étant alignés sur une ligne de cache, soit 128 octets pour le premier niveau, et regroupés par *warp* (voir le paragraphe 3.2), il faut alors s'assurer que les *threads* du même *warp* accèderont aux cellules successives de la base de données, et ceci grâce aux variables internes (voir la figure 3.4) qui indexent les *threads*. Le contrôleur mémoire du GPU s'arrangera alors pour assembler les accès à des espaces adjacents. On crée donc une fonction qui lira la liste de signatures à rechercher, et qui fournira la référence des paquets contenant les données incriminées afin d'en retracer

ensuite la source (IP et port des deux machines concernées).

Notons qu'on peut ici tirer profit des trois dimensions des variables d'indexation des *threads*. On peut par exemple spécifier que la valeur selon `x` renvoie au numéro de la signature à chercher tandis que celle selon `y` indique la colonne de la base de données à utiliser dans la comparaison :

$$\text{colonneIdx} = \text{blockIdx.x} \times \text{gridDim.x} + \text{threadIdx.x} \tag{3.8}$$

$$\text{signatureIdx} = \text{blockIdx.y} \times \text{gridDim.y} + \text{threadIdx.y} \tag{3.9}$$

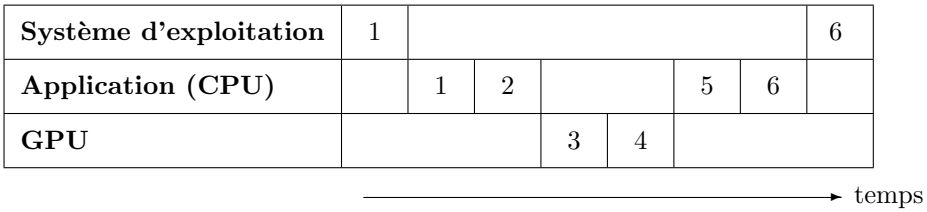
La décomposition influe par contre sur l'alignement des accès mémoire, car les *threads* sont groupés en *warp* selon l'index global (selon les trois coordonnées). On a donc intérêt à placer l'index de la colonne sur x pour que deux *threads* consécutifs accèdent à des adresses elles-aussi consécutives.

Pour réduire les données à transmettre sur le bus, nous avons créé un drapeau unique, que les *threads* doivent activer lorsqu'ils ont trouvé une correspondance. Le CPU n'a ainsi qu'à lire ce drapeau, et ne récupère l'ensemble des résultats incluant les positions des paquets incriminés dans le *buffer* que lorsque nécessaire. Notons que le transfert des résultats vers le CPU à la fin du traitement n'a pas d'impact sur cette bande passante puisque le bus PCIe est full-duplex, ce qui signifie que l'information est transportée simultanément dans les deux sens. De plus, les quantités de données transmises sont négligeables. Le principe est repris à la figure 3.14.

Données: les signatures N à chercher sig et leurs positions pos associées, la base de données ref de c colonnes et 2^p lignes
Résultat: drapeau $flag$ et tableau de N positions res_pos

$colonne \leftarrow index.x$
$sig\_ind \leftarrow index.y$
$ligne \leftarrow sig[sig\_ind]\%2^p$
si $ref[ligne][colonne] == sig[sig\_ind]$ **alors**
  $flag \leftarrow 1$
  $res\_pos[sig\_ind] \leftarrow res\_pos[sig_ind]$
sinon
  $res\_pos[sig\_ind] \leftarrow (-1)$
finsi

Figure 3.14 – Principe de la recherche des signatures



D'après les numéros d'étapes du paragraphe 3.4.1, et sans indication de durée des tâches.

Figure 3.15 – L'enchaînement complet des opérations d'analyse

3.4 Système complet

Nous disposons alors de trois fonctions principales pour calculer des signatures, les enregistrer, ou les chercher dans la base de données. À partir de cela et des fonctions d'écoute du réseau, nous pouvons obtenir une solution logicielle complète. Nous allons nous intéresser à l'enchaînement des actions par les différents processeurs, à l'utilisation de la mémoire, puis proposer des exemples de systèmes formés.

3.4.1 Répartition des tâches

Les étapes de l'analyse sont les suivantes (le référencement est équivalent), avec le "relai" entre CPU et GPU :

1. CPU : Récupération des paquets grâce à `pcap` dans un *buffer*
2. CPU : Envoi du *buffer* vers la mémoire dédiée au GPU
3. GPU : Calcul des signatures
4. GPU : Recherche des signatures
5. CPU : Envoi des résultats vers la mémoire centrale
6. CPU : Analyse des résultats et blocage en conséquence grâce aux `ebtables`

L'analyse des résultats est effectuée sur le CPU car elle n'est pas suffisamment adaptée au fonctionnement du processeur graphique. En effet, il s'agit de "compter" les alertes concernant une même connexion, et de communiquer avec le système d'exploitation pour la bloquer si nécessaire. On gardera une analyse séquentielle, exécutée uniquement lorsque le GPU a détecté une correspondance.

La figure 3.15 représente l'enchaînement des fonctions exécutées. On remarque vite sur la chronologie que chacun des composants passe beaucoup de temps en attente, ce qui renforce l'intérêt et le besoin de mettre en place un système avec un pipeline à plusieurs étages.

Pour ce faire, on pourra lancer plusieurs processus sur le CPU, d'autant que les processeurs récents sont largement multicœurs et pourront donc exécuter toutes les actions en parallèle, garantissant ainsi l'efficacité du pipeline et une plus grande bande passante des traitements. On trouve des processeurs à 6, voire 8 cœurs en parallèle, ce qui permet de lancer des applications très complexes.

La configuration des grilles de lancement des fonctions sur le GPU dépend bien sûr des données entrantes. Considérons un *buffer* de N octets, que l'on divise en B blocs avant de calculer quatre signatures à partir de chacun d'eux. On lance alors B *threads* pour le calcul les signatures, chacun assigné à un bloc de données. Ces *threads* vont chacun fournir quatre signatures. On lance à la suite $B \times c$ *threads* recherchant les signatures, chacun s'occupant d'une signature et une colonne de la base de données. On récupère alors un drapeau

3.4.2 Mémoire nécessaire

Chacune des fonctions doit travailler sur des données dans la mémoire globale, tout du moins pour les données sources/destinations, les intermédiaires pouvant être rapprochés dans les registres ou la mémoire partagée. On doit donc disposer de plusieurs espaces mémoire dans la mémoire globale de la carte graphique pour recevoir les paquets, enregistrer les signatures calculées puis les résultats de la recherche, et stocker la base de données. Pour une plus grande efficacité, on remplira un *buffer* pendant qu'un autre sera analysé. On doit alors diviser la mémoire disponible en au moins deux buffers de paquets, deux buffers de signatures, deux buffers de résultats et un base de données. Si l'on dispose de beaucoup de mémoire disponible, il peut être rentable de créer plus que deux espaces de travail pour garantir une bonne occupation du GPU même lorsque les *buffers* à traiter sont relativement petits.

3.4.3 Mise en place finale

Les fonctions sont codées en C++ pour la partie CPU, et en CUDA C pour la partie GPU. On dispose alors de toutes les fonctions nécessaires au bon fonctionnement du système : les outils de récupération des données, des *kernels* de calcul et de recherche des signatures, et une méthode de filtrage. Plusieurs agencements sont alors possibles, suivant le matériel disponible et les performances nécessaires, que nous allons illustrer par deux exemples.

Monoprocesseur

Le choix le plus économique paraît être de limiter le matériel à un CPU et un GPU. On lance alors successivement les fonctions de calcul des signatures, puis la recherche dans la

base de données. *A priori*, cette architecture est la moins efficace puisque le processeur doit exécuter plusieurs fonctions, soit en parallèle, soit les unes après les autres. Dans tous les cas, les fonctions doivent partager le temps disponible sur le processeur. De même, la mémoire doit être partagée entre la base de données et les emplacements en mémoire pour recevoir les données et les signatures calculées. Ce choix limite en revanche le nombre de transferts puisque seuls deux espaces mémoire se relaient dans le traitement de l'information.

Multiprocesseur

En multipliant les processeurs, on multiplie facilement le taux de calcul et donc la bande passante analysable en temps réel. Si cette architecture nécessite plus de transferts des données d'une carte à l'autre, chaque GPU peut être dédié à une fonction et donc beaucoup plus spécialisé. Par exemple, une carte graphique pourrait être dédiée à la recherche de signatures dans la base de données, et réserver ainsi l'ensemble de sa mémoire à ladite base de données et à un *buffer* de signatures à rechercher. Les limites sont alors imposées par la bande passante des transferts par les bus internes de la machine, et la latence du traitement.

CHAPITRE 4

ANALYSE DU SYSTÈME POUR LE PROBLÈME POSÉ

Nous avons présenté au chapitre 3 notre système d'analyse et de filtrage ainsi que ses différents composants. Il convient alors de mesurer les performances de son implémentation pour pouvoir valider les choix effectués ou envisager des modifications. Avec ces résultats, nous serons en mesure de présenter la meilleure réponse possible à chaque type de déploiement puisque nous connaîtrons les performances exactes que nous pourrons proposer pour chaque type d'implémentation. Nous commencerons donc par étudier les principales fonctions de l'algorithme à la section 4.1, puis la gestion du réseau à la section 4.2, avant de pouvoir présenter à la section 4.3 les choix réalisables suivant le cahier des charge imposé.

Notre banc de tests pour ces mesures est basé sur des serveurs constitués de deux processeurs Intel Xeon 5650 (12 cœurs au total), 64 Go de mémoire vive, un contrôleur Ethernet Intel 82599 avec deux ports Ethernet à 10 Gb/s et quatre cartes graphiques NVidia Tesla C2050. Ces cartes sont reliées au processeur et à la mémoire par un lien PCI Express version 2.0, avec 16 lignes (64 Gb/s théoriques) pour les cartes graphiques et 8 lignes (32 Gb/s théoriques) pour les cartes réseau.

Les données que nous utiliserons sont des données aléatoires, pour représenter statistiquement le trafic d'un réseau important et diversifié. Avant de mesurer le taux de traitement des fonctions sur le GPU, nous copierons ces données aléatoires dans la mémoire de la carte graphique afin de ne pas prendre en compte les transferts, qui seront mesurés séparément, dans ces tests.

Pour les tests de réseau, nous relierons trois serveurs en série comme illustré sur la figure 4.1 : le premier et le troisième communiqueront ensemble tandis que le second servira d'intermédiaire dans leur communication mais leur sera invisible. C'est sur cet ordinateur que nous analyserons et filtrerons le flux, avec éventuellement du trafic réel.

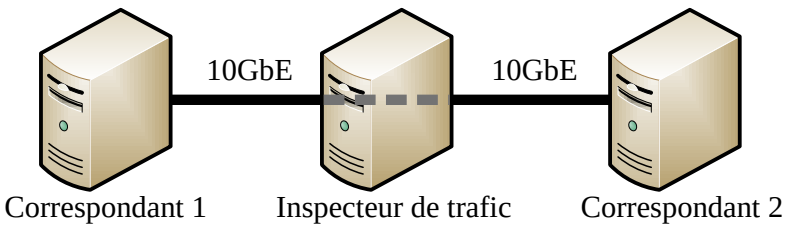


Figure 4.1 – Installation des serveurs pour les tests de réseau

4.1 Performances du GPU

Nous allons dans cette partie mesurer les performances des différents composants logiciels du système. Nous commencerons par analyser les taux de transferts atteignables sur le bus PCIe pour fixer la gamme de débits avec laquelle nous pourrons travailler. Nous nous intéresserons alors aux fonctions de calcul et de recherche des signatures.

4.1.1 Mesures de temps sur le GPU

Comme nous l'avons évoqué à la section 3.2, l'exécution des calculs sur le GPU est asynchrone, c'est-à-dire que l'appel d'un *kernel* retournera dès que le *driver* de la carte réseau aura lancé les *threads* correspondants, sans attendre que ceux-ci aient terminé leur travail. Le *driver* peut heureusement s'occuper de mesurer le temps effectivement passé dans le *kernel* par le GPU. Pour cela, il crée un *stream* et enregistre deux événements pour le début et la fin de l'exécution. Lorsque le *stream* est entièrement exécuté, il reste à mesurer la différence de temps entre les deux événements. Ce système peut donc être utilisé sur toutes les actions du GPU, en particulier les transferts et les lancements de fonctions. Un modèle est présenté sur la figure 4.2. On y observe la création des événements, ainsi que leur enregistrement aux instants requis. Ces instructions étant asynchrones, il faut finalement que tout soit terminé avant de calculer l'intervalle de temps recherché.

```
1  // Creation de deux evenements
2  cudaEvent_t debut, fin;
3  cudaEventCreate( &debut );
4  cudaEventCreate( &fin );
5
6  // On entoure la fonction de l'enregistrement des evenements
7  cudaEventRecord( debut, 0 );
8  ma_fonction <<< blocs_grille, threads_par_bloc >>> ( /* arguments */ );
9  cudaEventRecord( fin, 0 );
10
11 // attente que le second evenement soit termine
12 cudaEventSynchronize( fin );
13
14 // Calcul du temps ecoule entre debut et fin
15 float temps_ms;
16 cudaEventElapsedTime( &temps_ms, debut, fin );
```

Figure 4.2 – Mesure du temps sur le GPU

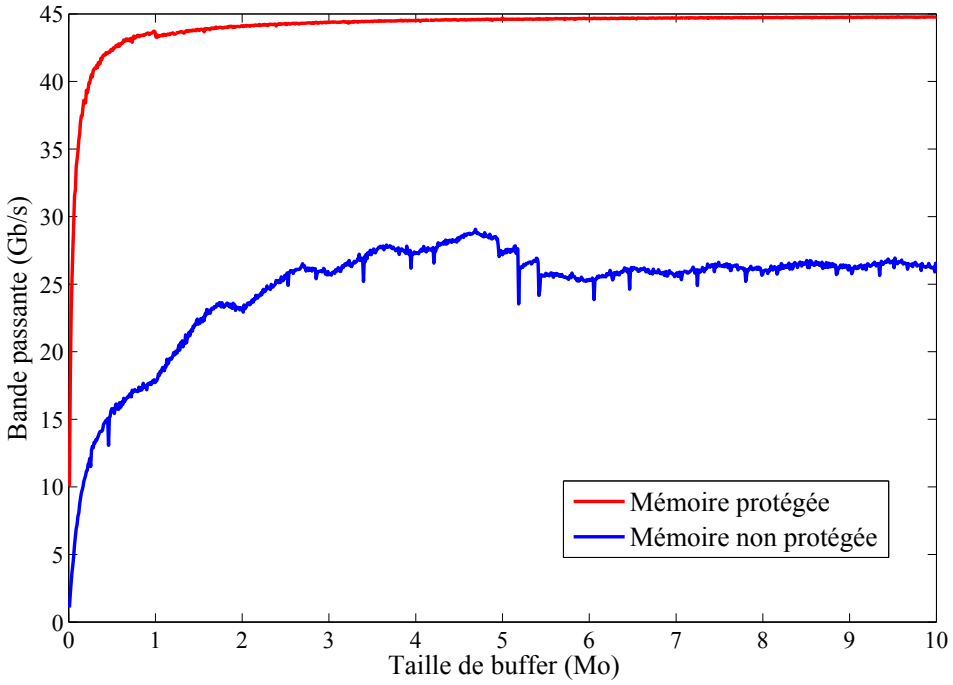


Figure 4.3 – Bande passante du PCI Express

4.1.2 Transferts mémoire

Pour amener les données de la mémoire centrale à la mémoire du GPU, on utilise le bus PCI Express en DMA (*Direct Memory Access*). Le taux de transfert théorique de notre configuration (PCIe v2.0 16x) est 64 Gb/s. On mesure en réalité un taux de transfert maximal de 44.96 Gb/s du CPU vers le GPU. Cette bande passante dépend de la quantité de données envoyées par transfert, mais on observe sur le graphique de la figure 4.3 que l'on atteint assez rapidement une vitesse proche du maximum pour des tailles de *buffer* de l'ordre de quelques mégaoctets. La bande passante dépend aussi du type d'emplacement mémoire "source". En effet, le DMA est le plus efficace lorsque l'espace mémoire à lire est protégé contre le *paging*, c'est-à-dire le déplacement de l'espace de la mémoire vive vers le disque dur, offrant plus de mémoire vive à un autre processus. La différence de bande passante est conséquente, comme on peut le voir sur la figure 4.3. Pour utiliser ce type de protection, dite "*page-lock*", on doit par contre s'assurer de laisser assez d'espace dans la mémoire vive pour le que le système d'exploitation et les autres logiciels puissent continuer de fonctionner normalement.

Ces résultats montrent clairement la limite de transfert du bus, mais nous confortent dans l'idée que nous pourrons travailler sur des débits de 40 Gb/s. Il faut alors analyser les fonctions de calcul pour s'assurer qu'elles peuvent traiter les données à ce rythme.

4.1.3 Calcul des signatures

Comme évoqué au paragraphe 3.3.1, la fonction de calcul des signatures est peu adaptée aux spécificités du GPU. Il convient d'y apporter un soin particulier puisque cette fonction effectuera tous les calculs de notre système. Il s'agira donc vraisemblablement de la principale limitation du débit traité.

Le chemin de données est le suivant : à chaque itération du déplacement de la fenêtre de calcul, l'octet entrant et l'octet sortant sont lus dans la mémoire globale pour adapter la fenêtre. On calcule alors sa signature et on met le maximum à jour. On calcule ainsi quatre signatures par bloc de données en enregistrant la position de chacune. Plusieurs améliorations successives montrent bien l'impact des différents modes d'accès à la mémoire globale, souvent le principal point limitant des applications GPU.

L'ensemble des résultats que nous présentons ici est repris dans le tableau 4.1. La bande passante traitée par le système dépend du nombre de signatures calculées et donc de la configuration choisie. On fournira ainsi le nombre de signatures calculées, qui est plus représentatif que la bande passante.

Accès à la mémoire

Les résultats mesurés en fonction des méthodes d'accès présentées dans le paragraphe 3.3.1 suivent bien nos attentes. La version simple, dans laquelle on lit chaque octet nécessaire ne peut traiter qu'une faible bande passante, il ne produit que 7.5×10^6 signatures par seconde. Lorsqu'on désactive la cache L1 en n'utilisant plus que la cache L2, les performances montrent un léger gain d'efficacité puisque l'on mesure alors au taux de 9.6×10^6 signatures produites par seconde. Ce faible gain montre pourtant qu'il ne s'agit pas de la principale limite.

Tableau 4.1 – Débit du calcul des signatures

Configuration	Débit (signatures par seconde)
Basique	7.5×10^6 sig/s
Sans cache L1	9.6×10^6 sig/s
Sauvegarde des octets lus	31.9×10^6 sig/s
Lecture par 16 octets	146.8×10^6 sig/s
Sauvegarde en registres	150.0×10^6 sig/s
Filtrage des octets entrants	96.5×10^6 sig/s

En sauvegardant les octets lus pour les réutiliser ensuite, les performances mesurées sont nettement plus importantes puisque le GPU calcule 31.9×10^6 signatures par seconde, malgré le rendement faible des lectures. On augmente alors la taille des lectures en sauvegardant temporairement les valeurs dans des registres. Les performances sont assez satisfaisantes puisque l'on calcule alors 146.8×10^6 signatures par seconde.

Lorsque l'on a déplacé le tableau de valeurs sauvegardées de la mémoire locale vers les registres, les performances ont gardé le même ordre de grandeur. Le processeur crée 150.0×10^6 signatures par seconde. Ceci montre que les accès à la mémoire locale sont convenablement optimisés. De plus, on arrive ici à une redistribution intéressante des performances : le compilateur optimise le *kernel* pour qu'il utilise moins de registres (36 au lieu de 50 précédemment) et accepte des accès à la mémoire sous-optimaux (40% des données lues sont utilisées). Malgré cette possible baisse de performances, le plus faible nombre de registres permet aux multiprocesseurs de gérer un plus grand nombre de *threads* en parallèle (voir le tableau 3.1 pour le nombre de registres disponibles), ce qui accélère globalement le traitement des données. Cette optimisation n'aura donc d'intérêt que si la quantité de données à traiter est suffisante. Si le nombre de *threads* lancés ne suffit pas à occuper les multiprocesseurs, il n'y aura pas assez de traitements pour masquer la moins bonne gestion des accès mémoire et on ne pourra pas bénéficier de performances intéressantes. On utilisera cette méthode en gardant cette limite à l'esprit, sachant que le retour à la méthode précédente n'impacte pas le reste des fonction et garde des performances du même ordre de grandeur.

Divergences

Dès lors que l'on veut ajouter des contrôles (filtres, mesures, ...), on oblige le compilateur à créer des divergences qui limitent beaucoup les performances du GPU. Nous avons par exemple implémenté une version de l'algorithme optimisée pour l'analyse de textes. Cette version utilise un filtrage simple sur les données entrantes pour supprimer les octets en double et les valeurs ASCII de contrôle (retour, nouveau paragraphe, etc.). On teste donc la valeur du nouvel octet, et le cas échéant on passe directement à la fenêtre suivante.

Les divergences créées font diminuer le taux de traitement de plus de 35%, passant de 150 à 96.5 millions de signatures calculées par seconde. Ces mesures illustrent très clairement l'intérêt de raffiner l'algorithme.

4.1.4 Recherche des signatures

La fonction de recherche des signatures est rudimentaire puisqu'elle se résume à une comparaison. En effet, on choisit de lancer un *thread* pour chaque signature recherchée et

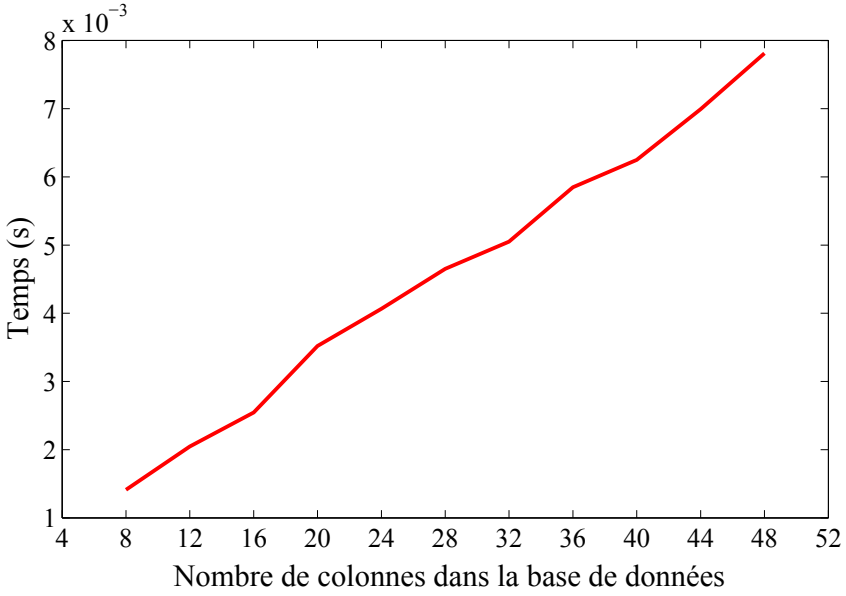


Figure 4.4 – Temps de recherche d'une signature

chaque colonne de la base de données. Les performances ne dépendent alors que du *pattern* utilisé pour accéder aux données. La lecture de la signature à chercher est commune pour autant de *threads* qu'il y a de colonnes dans la base de données. La lecture de la base de données est linéaire pour ces *threads* car les colonnes de la base de données sont consécutives, mais globalement aléatoires puisqu'on accède à la ligne indexée par les bits de poids faible des signatures.

Comme indiqué à la section 3.3.2 a choisi dans notre implémentation de regrouper en *warp* les *threads* qui accèdent à la même signature, pour que leurs accès à la base de données soient regroupés et plus rentables grâce à la cache. La valeur de la signature à chercher est transmise en même temps à tous les *threads* du *warp*, ce qui limite donc le nombre d'accès. Afin de pouvoir visualiser l'impact de l'alignement des accès mémoire, nous avons aussi utilisé un *pattern* non optimal dans lequel un *warp* accède à plusieurs signatures et à des colonnes quelconques de la base de données. Les résultats, repris dans le tableau 4.2, montrent une nette différence de performances.

Tableau 4.2 – Performances de la recherche de signatures

Configuration	Recherches	Comparaisons
Accès non optimisés	450×10^6 sig./s	3.6×10^9 comp./s
Accès optimisés	708×10^6 sig./s	5.7×10^9 comp./s

Ces tests sont réalisés avec une base de données de 16 colonnes, dans un cas plutôt défavorable : les *threads* accèdent à des lignes aléatoires de la base de données. En comparant les performances en fonction du nombre de colonnes de la base de données, on obtient un taux moyen constant de six milliards de comparaisons par seconde. On vérifie sur le graphique de la figure 4.4 que le temps pour rechercher une signature est proportionnel au nombre de colonnes dans la base de données, c'est-à-dire au nombre de comparaisons à réaliser. La vitesse de recherche des signatures pourra ainsi influencer le choix de la taille de la base de données puisque suivant l'architecture choisie, on aura besoin d'une vitesse plus ou moins importante. Ce choix ne contrôle pas directement la taille de la base de données, qui sera gérée par le nombre de lignes, avec lequel la vitesse de recherche est indépendante.

4.1.5 Fonctions enchaînées

Les deux mesures ci-dessus sont effectuées dans le cas où une fonction unique dispose de l'ensemble du processeur. Un test intéressant consiste à enchaîner les étapes de calcul et de recherche sur le même processeur, les faisant ainsi partager ses ressources. Le test le plus représentatif est l'utilisation d'un pipeline tel que celui de la figure 4.5. Le but est ici de traiter les données le plus rapidement possible. Les transferts sont physiquement limités et l'on souhaite que cette limite soit la seule. Le temps nécessaire pour les deux fonctions et le transfert des résultats doit ainsi être inférieur au temps de transfert des données à traiter.

Les mesures valident ce mode de fonctionnement, comme le montre la figure 4.5 avec l'exemple d'un *buffer* de 512 Mo. On peut donc potentiellement traiter près de 45 Gb/s de données avec une unique carte graphique. Nous présenterons les techniques pour travailler avec des débits plus importants au paragraphe 4.3.

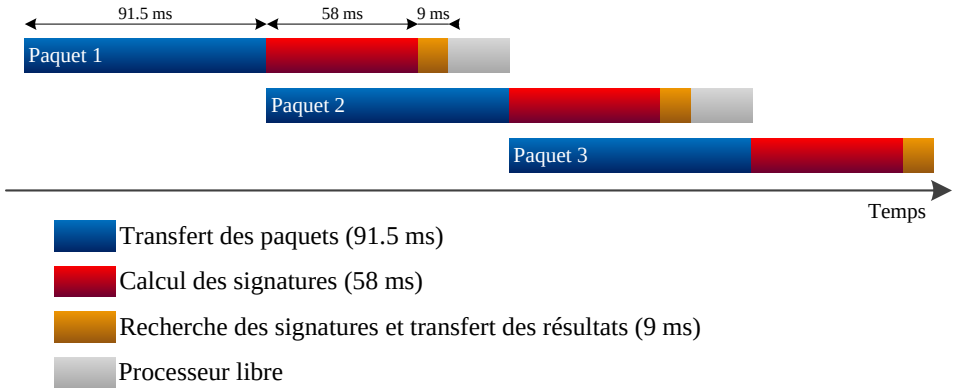


Figure 4.5 – Pipeline de l'application

4.2 Performances du réseau

Même si la partie du système qui nous concerne ici est le traitement des données par le processeur graphique, il est intéressant d'avoir une idée des capacités de traitement des flux réseau par le système d'exploitation, Linux dans notre cas. Nous l'utilisons pour créer un pont réseau entre deux cartes Ethernet, filtrer le trafic selon les règles des `ebtables` présentées à la section 3.1.3, et copier le trafic restant pour le transmettre à notre application (voir section 3.1.1). Nous allons étudier ici les performances sur un réseau 10 Gb/s pour donner un aperçu des avantages et inconvénients de cette méthode.

4.2.1 Pont réseau

Pour mesurer la capacité de Linux à gérer un pont entre deux réseaux, nous avons mis en place deux serveurs communiquant entre eux et générant ainsi du trafic à près de 10 Gb/s. Nous avons mesuré le débit moyen des données transmises avec et sans serveur intermédiaire. Les résultats sont repris dans le tableau 4.3 et montrent que la gestion par Linux est assez efficace pour interconnecter deux réseaux sans causer de ralentissement. On observe une légère augmentation de la latence.

Le tableau montre également que le système d'exploitation assigne une priorité très élevée au pont réseau car les performances restent identiques lorsque le processeur est très occupé. On visualise l'impact sur le processeur en suivant l'activité de programme qui chargent les cœurs du processeur à 100%. En effet, lorsque du trafic réseau doit être traité, les programmes sont préemptés, ne disposant plus que de 86% du temps CPU. Ceci tend à montrer qu'à débit maximal, la gestion du pont réseau seule utilise 14% de la puissance de calcul des processeurs, ce qui était prévisible puisque tout est géré en logiciel.

Ici revient l'intérêt d'utiliser un périphérique dédié et spécialisé, pour gérer le réseau et décharger le processeur de cette tâche. Les cartes réseau haut de gamme peuvent même offrir

Tableau 4.3 – Performances du pont réseau

État du serveur intermédiaire	Débit moyen	Ping moyen
Référence : connexion directe	8.85 Gb/s	0.1 ms
Libre (charge $\ll 0.1\%$)	8.85 Gb/s	0.3 ms
Occupé (charge 100%)	8.85 Gb/s	0.2 ms
Avec copie du trafic	8.85 Gb/s	0.3 ms
Filtrage (100 règles)	8.85 Gb/s	0.3 ms
Filtrage (32 762 règles)	8.85 Gb/s	0.3 ms

cette fonction en matériel, ne nécessitant alors pas de périphériques externes.

4.2.2 Copie du trafic

En plus de simplement relier les deux ports Ethernet, le système d'exploitation doit aussi passer les paquets de données à notre application d'analyse, ce qui charge encore le processeur. Nous avons repris la même plateforme de test que précédemment en ajoutant le processus de copie des paquets vers un *buffer* dédié à une application.

Notons que la copie des données depuis la mémoire de la carte réseau jusqu'à la mémoire de la carte graphique passe par le bus PCIe. Il serait envisageable de les faire communiquer directement en DMA, ne passant alors pas du tout par la mémoire vive du CPU. Malheureusement NVidia ne donne ni accès à la gestion du bus, ni directement à la mémoire de ses cartes, et nous sommes donc obligés de travailler avec la mémoire centrale comme intermédiaire.

La copie des données depuis la carte réseau que nous avons présenté au paragraphe 4.1.2 peut, elle aussi, être réalisée selon différentes méthodes. En effet, par défaut, le système d'exploitation lit les paquets et les copie dans sa mémoire (qui lui est réservée et n'est pas accessible aux applications). On utilise alors des applications telles que *pcap* pour récupérer une copie de cette mémoire dans l'espace utilisable par l'application. Il existe cependant des alternatives telles que `ntop` [17], qui fonctionnent de la même façon que *pcap*, mais directement avec le driver de la carte réseau. Les paquets peuvent alors être copiés directement vers l'application, allant même jusqu'à outrepasser le système d'exploitation, qui ne voit donc plus de trafic. Il faut alors être sûr de pouvoir gérer le pont réseau et le filtrage autrement. Ces solutions peuvent être beaucoup plus efficaces car plus directes, mais demandent aussi un plus grand travail de mise en place pour des fonctions que le système d'exploitation fournissait très simplement. Des tests effectués sur une plateforme équivalente à la nôtre ont permis de récupérer tout le trafic à 10 Gb/s [16].

Analyser les possibilités de notre matériel et les alternatives à *pcap* après avoir optimisé la partie GPU serait très intéressant. En utilisant la librairie *pcap*, on parvient à récupérer l'ensemble du trafic jusqu'à 7.3 Gb/s.

4.2.3 Filtrage

Comme nous l'avons expliqué, le filtrage est effectué par le système d'exploitation et charge le processeur. Les essais montrent en effet l'impact des *ebtables* sur les performances du pont réseau, les résultats sont ajoutés au tableau 4.3.

Le filtrage peut être caractérisé par différents critères tels que le nombre de règles sup-

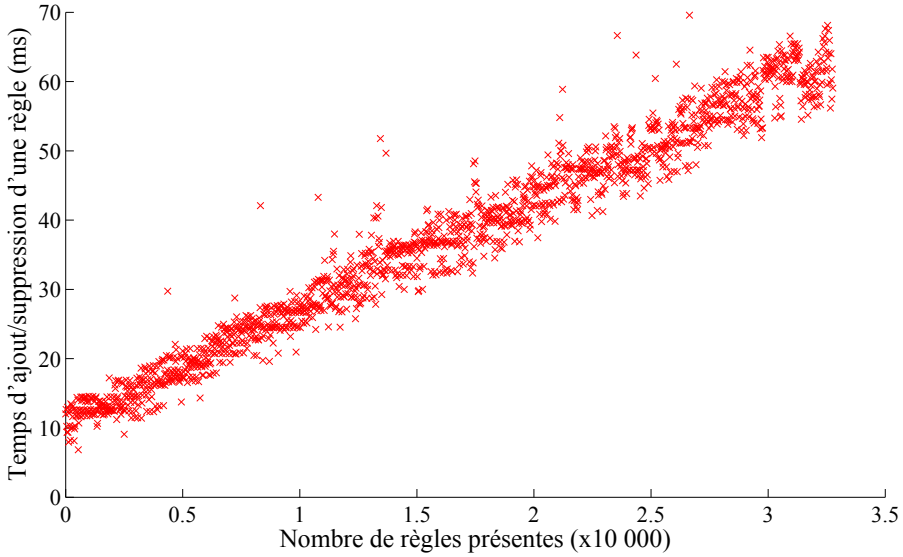


Figure 4.6 – Latence des ebtables

portées, le mode d'ajout ou encore la latence entre la demande d'ajout et la mise en place effective. Les *ebtables* peuvent prendre en compte 32 762 règles sur notre système de test. On mesure que la latence de l'ajout ou la suppression d'une règle est globalement proportionnelle au nombre de règles existantes, comme le montre la figure 4.6. Cette latence est assez importante, entre 5 ms et 70 ms. On ne pourra donc utiliser efficacement ce système qu'avec un petit nombre de règles de blocage.

Minimiser la latence entre l'arrivée d'un paquet et la mise en place de la règle de filtrage correspondante est en effet primordial. Nous bloquons une communication à partir du moment où des données interdites ont été repérées à l'intérieur. En travaillant à de très hauts débits, chaque cycle passé dans le traitement retarde l'application du blocage et laisse l'utilisateur transférer beaucoup de données. Si le traitement est trop long, le transfert du fichier peut être terminé avant que le blocage n'ait été mis en place.

Nos tests de filtrage ont montré une latence comprise entre 7 et 40 ms, l'écart entre les mesures étant du au temps de remplissage du *buffer* d'entrée. Avec de telles latences, on peut aisément bloquer les communications classiques (chargement d'images, lecture de vidéos) sur Internet, les utilisateurs ne pouvant alors télécharger leur contenu que pendant les quelques millisecondes du traitement du premier paquet suspect.

4.3 Choix de l'architecture

Nous avons maintenant une idée précise des possibilités de notre système d'analyse et de filtrage. Il est donc possible de créer des systèmes répondant à chaque type de besoin. Nous allons détailler ici quelles en sont les configurations possibles.

4.3.1 Traitement à haute vitesse

Comme nous l'avons signalé plus haut, un seul GPU peut traiter les données entrantes à près de 45 Gb/s. Pour des débits plus importants, des approches à plusieurs processeurs peuvent être mises en places, comme illustré sur la figure 4.7, le bus restant la seule limite pour apporter les données jusqu'au processeur. En effet, le nombre de liens PCIe et leur répartition dépend du CPU et du contrôleur d'entrées/sorties auquel il est relié. Les processeurs récents peuvent fournir 40 liens et les cartes graphiques en utilisent 16. On pourra alors répartir les cartes graphiques de façon adéquate pour qu'elles puissent toutes obtenir le plus grand débit de données possible. Dans ces conditions, on multiplie directement le débit de traitement par le nombre de cartes graphiques.

Le calcul des signatures demande aussi plus de temps que la recherche de celles-ci. Il pourrait être intéressant de dédier un GPU à la recherche dans la base de données, lui réservant en même temps toute la mémoire de la carte graphique. Des applications demandant de repérer particulièrement beaucoup de documents originaux pourront préférer une telle approche. Dans la configuration testée à la section 4.1, on pourrait utiliser une carte

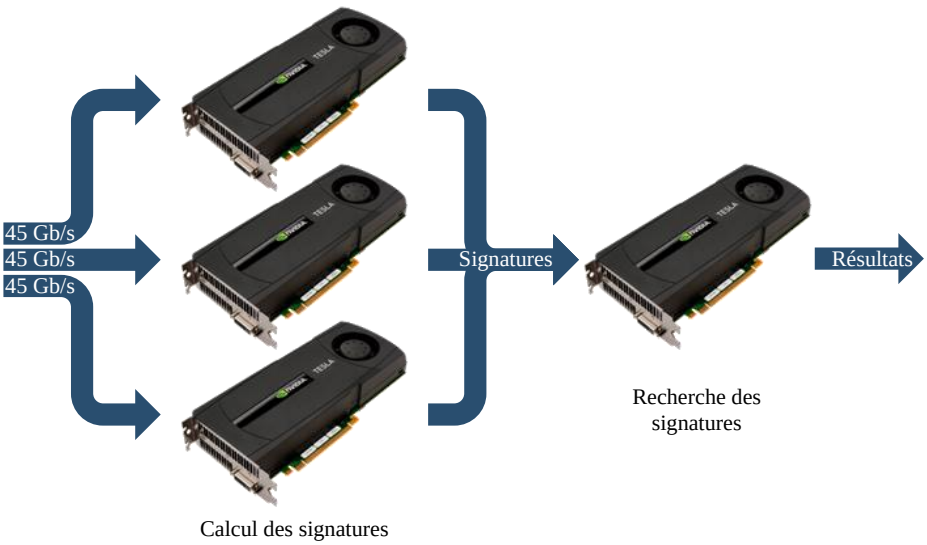


Figure 4.7 – Exemple d'agencement multiprocesseur

graphique avec une mémoire plus importante pour rechercher les signatures créées par cinq autres cartes moins puissantes fonctionnant à plein débit. On parvient ainsi à optimiser la quantité de signatures stockées tout en optimisant les coûts.

4.3.2 Acquisition et filtrage du trafic

Le plus gros challenge reste la gestion du réseau. Si les optimisations que nous avons évoquées précédemment peuvent apporter beaucoup, notamment au niveau de la charge des CPU, la solution la plus viable pour une augmentation conséquentes des débits est d'assigner cette gestion à un périphérique externe, qui sera basé sur un processeur réseau et se chargera de gérer le pont réseau ainsi que le filtrage au niveau du TCP/UDP. Une carte réseau haut de gamme pourra notamment être reliée au bus PCI Express et fournir directement les données dans les espaces mémoire dédiés à l'application, le système d'exploitation n'intervenant que très peu. Notons que dans ce cas, le système d'exploitation n'a même plus besoin de recevoir une copie des paquets puisqu'il n'a rien à gérer au niveau de ce réseau. Il se chargera donc uniquement de la configuration des DMA et de la répartition des tâches.

CHAPITRE 5

CONCLUSION

5.1 Synthèse des travaux

Nous avons proposé dans ce mémoire un système de détection de contenus connus au sein de communications à très haut débit. L'algorithme de *max-hashing* a servi de base à ce travail, et a été implémenté et optimisé pour les processeurs graphiques (GPU). Cet algorithme est spécialisé dans la détection rapide de contenus connus à partir de fragments. Il est donc idéal pour l'analyse de transmissions Ethernet dans lesquelles les documents transmis sont divisés en paquets de quelques centaines d'octets. L'implémentation sur GPU est motivée par l'importante puissance de calcul qui rend ce type de processeur idéal pour traiter le flux à un débit très élevé. Le parallélisme intrinsèque est mis à contribution pour traiter indépendamment chaque paquet Ethernet.

Les fonctions exécutées sur le GPU ne peuvent accéder qu'à la mémoire embarquée sur la carte graphique. Il faut donc d'abord y copier les paquets reçus du réseau. Le débit de données maximal qu'une carte peut avoir à traiter est donc limité par le bus PCI Express qui la relie au CPU. On mesure une bande passante maximale de 45 Gb/s environ (PCIe v2.0 16x). L'implémentation de l'algorithme a donc pour objectif de traiter ce même taux de transfert de données.

L'algorithme utilise des signatures pour référencer les documents qui seront filtrés, ce qui est beaucoup plus efficace à traiter qu'un document quelconque et présente l'avantage de ne pas donner accès aux documents sensibles dont on veut précisément limiter la publication. Ces signatures sont stockées dans une base de données, elle-même embarquée dans la mémoire de la carte graphique pour des accès rapides depuis le GPU. Lorsqu'on souhaite analyser un paquet IP, on en lit le contenu à partir duquel on calcule des signatures. Il suffit alors de chercher si des correspondances existent dans la base de données.

Notre implémentation peut, sur une carte graphique unique, extraire 150 millions de signatures par seconde ou effectuer 5.7 milliards de comparaisons dans la base de données par seconde. On peut alors ajuster précisément les performances de chaque étape du traitement. En configurant le nombre de signatures calculées sur chaque ensemble de données, on définit la bande passante admissible par le système. On peut aussi définir la taille de la base de données en fonction du nombre de signatures à sauvegarder et du nombre de comparaisons que l'on a le temps d'effectuer pour chaque signature recherchée. Les tests dans un cas très

défavorable où l'on calcule quatre signatures par bloc de 256 octets montrent qu'une unique carte graphique traite plus de 45 Gb/s de données sources. Un pipeline est mis en place entre les transferts et les calculs, de sorte que le taux de traitement global soit finalement fixé à la bande passante du PCI Express. En utilisant plusieurs cartes en parallèle on peut alors multiplier la bande passante admissible. Des choix avisés du matériel utilisé permettent de traiter de très hauts débits tout en optimisant les coûts, notamment par la spécialisation de chacune des cartes.

Ce système permet, à la différence des systèmes communs basés sur des expressions régulières, de repérer plusieurs centaines de millions de signatures différentes, sans subir de ralentissement. Cette capacité, alliée à la gamme de débits traitée et au faible coût du matériel, font de notre travail un système valable pour des FAI ou de grandes entreprises, qui pourront filtrer leurs réseaux à une très grande échelle sans impact sur ces réseaux.

Nous avons en outre pu créer un exemple de système complet de filtrage grâce aux fonctions de gestion du réseau du système d'exploitation Linux. L'interconnexion de deux ports Ethernet est gérée très simplement, ce qui permet d'installer le système de façon transparente dans un réseau existant. Linux propose aussi une solution simple de filtrage du trafic au niveau de la couche transport et en particulier des protocoles TCP et UDP. Ce filtrage peut être configuré rapidement et à la volée à partir des résultats de la détection. On est donc capable de bloquer la transmission de n'importe quel document en fermant la connexion correspondante dès que l'on détecte un paquet contenant une partie de ce document.

5.2 Limitations de la solution proposée

Si la partie de détection est très efficace, la principale limitation du système est la capture du trafic réseau. En effet, plusieurs copies des paquets sont nécessaires avec notre implémentation entièrement logicielle, deux au minimum (de la carte réseau vers la mémoire centrale, puis de la mémoire centrale vers la carte graphique) du fait de l'inaccessibilité de la mémoire de la carte graphique. Si les solutions étudiées dans ce travail permettent de gérer les 10 Gb/s de trafic de notre réseau de test, il paraît difficile d'envisager un trafic plus important sans déporter cette tâche sur du matériel spécialisé.

Pour traiter des débits de 40 Gb/s et plus, il faudra donc utiliser des cartes réseaux évoluées qui autoriseront la transmission directe des paquets vers la carte graphique ou au moins vers la mémoire dédiée au DMA du GPU. Les cartes réseaux évoluées permettent de gérer cela matériellement, ce qui décharge alors complètement le système d'exploitation et CPU.

5.3 Améliorations futures

Les futures étapes de ce travail iront donc dans ce sens. En effet, du nouveau matériel réseau devrait être testé par la suite, permettant de travailler à 40 Gb/s dans un premier temps. Du nouveau matériel disposant de PCI Express v3.0 pourra de plus faire transiter largement 100 Gb/s, et les nouveaux processeurs graphiques compatibles avec cette version du bus commencent à arriver sur le marché. Il est donc tout à fait envisageable de pouvoir travailler à 100 Gb/s dans un avenir proche. La comparaison des coûts avec les FPGA ou les processeurs réseaux utilisés généralement à de tels débits montre que notre système est une alternative performante et extrêmement économique.

5.4 Neutralité

Le système que nous avons élaboré au long de ce mémoire soulève plusieurs questions. La première concerne la protection de la vie privée des utilisateurs du réseau surveillé, puisque l'on a la capacité de lire le contenu de leurs communications. De plus, le filtrage proposé lors de la détection de données indésirables pose le problème de la définition de ces indésirables et de leur disponibilité au sein de la mémoire du système. Nous allons dans cette section discuter ces deux questions avant d'établir une liste de pratiques qui nous semblent indispensable pour conserver des réseaux de qualité. L'objectif de cette section est de susciter une discussion et des réflexions entre les acteurs qui utiliseront le système.

5.4.1 Protection de la vie privée

Dès lors qu'un système est capable "d'écouter" une communication tierce, on s'expose à des risques de mauvaise utilisation virant rapidement à la surveillance ou l'espionnage d'autrui. Nul ne peut s'arroger le droit de contrôler les communications d'un individu sans même que celui-ci soit au courant. De plus, personne ne peut garantir que le message traité ne contiendra pas d'informations hautement confidentielles qu'un correspondant peu au fait des risques aurait oublié de crypter.

L'un des risques couramment évoqué dans cette idée est la catégorisation des personnes. En effet, il est simple de reconstituer les usages que chaque intervenant fait du réseau, en associant son adresse IP avec le contenu des paquets correspondants. En cela, on commence très rapidement à classer les personnes selon différents critères. C'est la dérive classique de ce type de systèmes, qui prend la forme d'une optimisation de la gestion du trafic et de la qualité de service. De plus en plus de système impliquant des méthodes de *Deep Packet Inspection* sont ainsi mis en place, notamment au niveau des FAI qui affirment vouloir offrir

aux utilisateurs la “meilleure expérience possible” en adaptant très précisément le réseau à leur utilisation.

La protection de la vie privée est au cœur de nombreuses polémiques dans différentes régions du monde. Si ces entreprises se soumettent généralement (au moins publiquement) aux lois et requêtes des gouvernements et des clients, la question de fond reste toujours la même : que se passera-t-il le jour où un dirigeant moins regardant acceptera de vendre les profils très précis des utilisateurs non plus de façon anonyme comme c'est le cas aujourd'hui, mais de façon nominative ? Les utilisations qui pourront être faites de ces données sont très diverses : profilage à l'embauche, restriction de l'accès aux assurances ou à divers services en fonction de l'historique des pages Internet consultées... Le lecteur intéressé pourra consulter l'excellent article de Gallagher [24] qui reprend ces considérations de façon très précise et objective.

La capacité de lecture laissée aux machines ne pose pas de problème en elle-même, mais il faut signaler que ces systèmes sont souvent associés à de l'archivage, de la surveillance, et offrent des accès de contrôle aux administrateurs. Sans considérer la divulgation de données directement par les personnes en charge de la maintenance, on ne peut négliger le vol d'informations personnelles par des pirates informatiques, ni l'accès presque direct à nos communications qui pourra être laissé à des services de police en cas d'investigations, à l'instar des écoutes téléphoniques ou le suivi des transactions d'une carte bancaire. Internet est utilisé très largement, et l'envie de pouvoir le contrôler est logiquement omniprésente.

Divers éléments imposent quelques règles lors de la création et l'utilisation de systèmes d'analyse, en plus des réglementations légales de beaucoup de pays limitant les abus. La machine traitant les données ne devrait par exemple jamais sauvegarder ou copier les données qu'elle analyse. L'accès aux journaux d'activités, qui pourront par exemple contenir les adresse IP impliquées dans les événements, devrait être strictement restreint et le stockage hautement sécurisé. D'autre part, on l'a signalé précédemment, la base de données contenant les fichiers sensibles ne devrait pas contenir ces fichiers directement mais leurs signatures par exemple, ou toute autre représentation illisible et ne pouvant pas être décodée.

5.4.2 Filtrage ou censure ?

Le système que nous avons présenté dans ce mémoire est prévu pour être lié à un mécanisme de blocage des communications. Dès que la transmission d'un contenu illicite est détectée, on termine la connexion correspondante.

La principale discussion qui doit avoir lieu lorsque l'on utilise ce type d'approche est : qui décide quel document interdire ?

Une discrimination de l'accès à Internet par rapport aux sites visités ou à l'utilisateur

concerné est simple à mettre en place. L'histoire comprend plusieurs exemples dans différents pays du monde, toujours à partir de la protection contre la pornographie infantile. Un rapport [19] publié lors de débats sur la légalisation de systèmes de filtrage en France affirme :

> "À ce jour, toutes les listes de sites filtrés par les pays qui ont mis en place des mesures de filtrage ont fini par être rendues publiques. Toutes ont montré que les filtres était appliqués à bien plus que des contenus pédophiles, le record à cette date étant détenu par l'Australie, dont les sites à contenus pédophiles ne représentent que 32% des sites filtrés, et ce avant même que l'Australie étende ses mesures de filtrages à d'autres types de contenus."

Si des entorses à la neutralité du réseau peuvent être justifiées par le besoin de décharger un réseau, ou par des menaces de sécurité, il est important pour éviter de telles dérives qu'elles soient appliquées temporairement uniquement, et dans la transparence, après discussion et validation par des experts.

5.4.3 Risques inhérents à notre système

Le système que nous avons mis en place au cours de ce mémoire n'enregistre jamais le trafic réseau ailleurs que dans sa mémoire vive, où sa durée de vie est très courte (le temps de l'analyse). Il pourra donc être utilisé en l'état pour capturer les communications. Des données personnelles sont néanmoins mises en jeu : les adresses IP des utilisateurs et des sites interdits qu'ils ont consultés. Ces données pourraient leur causer préjudice. Les rapports d'activité peuvent être amenés à enregistrer ces informations en cas de détection/alerte, et devront donc être stockés en sécurité pour prévenir les fuites de données et il devront surtout être détruits lorsqu'ils ne seront plus nécessaires.

D'autre part, la base de données ne contient aucun contenu lisible. Ceci est un avantage car même en cas de brèche dans la sécurité donnant accès à ces données, on ne pourra pas en divulguer la source. La méthode de création des signatures garantit en outre que même si l'on parvient à décoder la signature, ce qui revient à trouver les paramètres utilisés lors de leur création, on n'obtiendra qu'un extrait de 128 octets du document original. L'inconvénient est par contre que les signatures de référence peuvent provenir d'un organisme tiers, dans l'idéal de la justice, qui peut alors inclure n'importe quel document dans la liste.

Nous avons donc proposé un système de détection et filtrage de qualité, mais qui devra être utilisé avec les précautions qui s'imposent.

BIBLIOGRAPHIE

[1] Aho, A. V. (1990). Algorithms for finding patterns in strings. *Handbook of Theoretical Computer Science : Algorithms and complexity*, 1:255–300.

[2] Aho, A. V. et Corasick, M. J. (1975). Efficient string matching : an aid to bibliographic search. *Commun. ACM*, 18(6):333–340.

[3] Baeza-Yates, R. A. (1989). Algorithms for string searching. *SIGIR Forum*, 23(3-4):34–58.

[4] Barth, G. (1984). An analytical comparison of two string searching algorithms. *Information Processing Letters*, 18(5):249–256.

[5] Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426.

[6] Boyer, R. S. et Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772.

[7] Broder, A. et Mitzenmacher, M. (2004). Network applications of bloom filters : A survey. *Internet Mathematics*, 1(4):485–509.

[8] Callado, A., Kamienski, C., Szabo, G., Gero, B., Kelner, J., Fernandes, S. et Sadok, D. (2009). A survey on internet traffic identification. *Communications Surveys & Tutorials, IEEE*, 11(3):37–52.

[9] Chaudhary, A. et Sardana, A. (2011). Software based implementation methodologies for deep packet inspection. *In International Conference on Information Science and Applications (ICISA), 2011*, pages 1–10.

[10] Cho, Y. H., Navab, S. et Mangione-Smith, W. H. (2002). Specialized hardware for deep network packet filtering. *In Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 452–461, 740405. Springer-Verlag.

[11] Cisco (2012). Cisco visual networking index : Forecast and methodology, 2011–2016. *White Papers*, pages 1–16.

[12] ClamAV (2002). Clam antivirus. Tiré de `http://www.clamav.net`, consulté le 26 juillet 2012.

[13] Cloud, S. (2012). Spam. Tiré de `http://www.symanteccloud.com/globalthreats/`, consulté le 1er octobre 2012.

[14] Cogent Communications (2012). Network map. Consulté en ligne le 8 août à l'adresse `http://www.cogentco.com/network/`.

[15] David, J. P. (2009). File presence detection and monitoring. 12/995,337.

[16] Deri, L., Gasparakis, J., Waskiewicz, Peter, J. et Fusco, F. (2011). *Wire-Speed Hardware-Assisted Traffic Filtering with Mainstream Network Adapters*, chapitre 5, pages 71–86. Springer US.

[17] Deri, L. et Suin, S. (1999). *Ntop : Beyond Ping and Traceroute Active Technologies for Network and Service Management*, volume 1700 de *Lecture Notes in Computer Science*, pages 71–71. Springer Berlin / Heidelberg.

[18] Elmagarmid, A. K., Ipeirotis, P. G. et Verykios, V. S. (2007). Duplicate record detection : A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16.

[19] Epelboin, F. (2010). *Pédopornographie sur internet : le mensonge qui cache la censure 2.0. Wikileaks dynamite la Loppsi*. ReadWriteWeb.

[20] Erdogan, O. et Cao, P. (2007). Hash-av : Fast virus signature scanning by cache-resident filters. *International Journal of Security and Networks*, 2(1/2):50–59.

[21] Fechner, B. (2010). GPU-based parallel signature scanning and hash generation. *23rd International Conference on Architecture of Computing Systems (ARCS), 2010*, pages 1–6.

[22] Fowers, J., Brown, G., Cooke, P. et Stitt, G. (2012). A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. *In Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 47–56, Monterey, California, USA. ACM.

[23] GadAllah, S. M. (2003). The importance of logging and traffic monitoring for information security. *SANS InfoSec Reading Room*, Logging Technology and Techniques.

[24] Gallagher, S. (2012). Big brother on a budget : How internet surveillance got so cheap. Tiré de http://arstechnica.com/information-technology/2012/08/, consulté le 13 septembre 2012.

[25] Garfinkel, S., Nelson, A., White, D. et Roussev, V. (2010). Using purpose-built functions and block hashes to enable small block and sub-file forensics. *Digital Investigation : The International Journal of Digital Forensics and Incident Response*, 7:S13–S23.

[26] Gokhale, M., Dubois, D., Dubois, A., Boorman, M., Poole, S. et Hogsett, V. (2002). Granidt : Towards gigabit rate network intrusion detection technology. *In Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 404–413, 740231. Springer-Verlag.

[27] Google (2007). Safe browsing. https://developers.google.com/safe-browsing/, consulté le 26 juillet 2012.

[28] Horspool, R. N. (1980). Practical fast searching in strings. *Software : Practice and Experience*, 10(6):501–506.

[29] Hunt, J. W. et McIlroy, M. D. (1976). An algorithm for differential file comparison. *Computing Science Technical Report*.

[30] Hutchings, B. L., Franklin, R. et Carver, D. (2002). Assisting network intrusion detection with reconfigurable hardware. *In Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 111, 795954. IEEE Computer Society.

[31] IANA (2012). Service name and transport protocol port number registry. Tiré de http://www.iana.org/assignments/service-names-port-numbers/, consulté le 1er octobre 2012.

[32] IBM (2011). Bringing smarter computing to big data. *Smarter Planet*.

[33] IEEE (2008). IEEE 802.3™ : IEEE standard for local and metropolitan area networks. Tiré de http://standards.ieee.org/about/get/802/802.3.html, consulté le 7 août 2012.

[34] Jacob, N. et Brodley, C. (2006). Offloading IDS computation to the GPU. *In Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 371–380, 1191892. IEEE Computer Society.

[35] Jiangfeng, P., Hu, C. et Shaohuai, S. (2010). The GPU-based string matching system in advanced AC algorithm. *In IEEE 10th International Conference on Computer and Information Technology (CIT), 2010*, pages 1158–1163.

[36] Kakuru, S. (2011). Behavior based network traffic analysis tool. *In IEEE 3rd International Conference on Communication Software and Networks (ICCSN), 2011*, pages 649–652.

[37] Katashita, T., Yamaguchi, Y., Maeda, A. et Toda, K. (2007). Fpga-based intrusion detection system for 10 gigabit ethernet. *IEICE - Transactions on Information and Systems*, E90-D(12):1923–1931.

[38] Kemmerer, R. A. et Vigna, G. (2002). Intrusion detection : a brief history and overview. *Computer*, 35(4):27–30.

[39] Khronos Group (2012). Opencl official webpage. Consulté en ligne le 13 août 2012 à l'adresse `http://www.khronos.org/opencl/`.

[40] Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. *Automata Studies, Annals of Mathematics Studies*, 34.

[41] Knuth, D., Morris, Jr., J. et Pratt, V. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350.

[42] Kornblum, J. (2006). Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3, Supplement:91–97.

[43] Lin, Y.-D., Lin, P.-C., Lai, Y.-C. et Liu, T.-Y. (2009). Hardware-software codesign for high-speed signature-based virus scanning. *IEEE Micro*, 29(5):56–65.

[44] Liu, C.-H., Chien, L.-S., Chang, S.-C. et Hon, W.-K. (2012). PFAC library : GPU-based string matching algorithm. *In GPU Technology Conference 2012*.

[45] Long, C. et Guoyin, W. (2008). An efficient piecewise hashing method for computer forensics. *In First International Workshop on Knowledge Discovery and Data Mining, 2008. WKDD 2008.*, pages 635–638.

[46] Madani, A., Rezayi, S. et Gharaee, H. (2011). Log management comprehensive architecture in security operation center (soc). *In International Conference on Computational Aspects of Social Networks (CASoN), 2011*, pages 284–289.

[47] McCulloch, W. et Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5(4):115–133.

[48] McHugh, J., Christie, A. et Allen, J. (2000). Defending yourself : The role of intrusion detection systems. *IEEE Software*, 17(5):42–51.

[49] Merkle, R. C. (1988). A digital signature based on a conventional encryption function. *In A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, pages 369–378. Springer-Verlag.

[50] Michailidis, P. D. et Margaritis, K. G. (2000). String matching algorithms : Survey and experimental results. *International Journal of Computer Mathematics*, 76:411–434.

[51] National Institute of Standards and Technology (2003). National software reference library. Tiré de `http://www.nsrl.nist.gov`, consulté le 21 avril 2012.

[52] Navarro, G. (2001). A guided tour to approximate string matching. *ACM Computing Surveys (CSUR)*, 33(1):31–88.

[53] NetFilter (2011). ebtables. Tiré de `http://ebtables.sourceforge.net/`, consulté le 11 août 2012.

[54] NetFilter (2012). iptables. Tiré de `http://netfilter.org/projects/iptables/`, consulté le 11 août 2012.

[55] NVIDIA (2012). CUDA C Programming Guide version 4.2.

[56] PCI-SIG (2012). Pcie® base 3.0 specification.
Tiré de `http://www.pcisig.com/specifications/pciexpress/base3/`, consulté le 15 octobre 2012.

[57] Ponec, M., Giura, P., Wein, J. et Brönnimann, H. (2010). New payload attribution methods for network forensic investigations. *ACM Transactions on Information and System Security (TISSEC)*, 13(2):1–32.

[58] Qadeer, M. A., Zahid, M., Iqbal, A. et Siddiqui, M. R. (2010). Network traffic analysis and intrusion detection using packet sniffer. *In Second International Conference on Communication Software and Networks, 2010. ICCSN '10.*, pages 313–317.

[59] Rabin, M. O. (1981). Fingerprinting by random polynomials. Rapport technique 15-81, Center for Research in Computing Technology, Harvard University.

[60] Rabin, M. O. et Scott, D. (1959). Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125.

[61] Roberts, P. (2005). Dod seized 60tb in search for iraq battle plan leak. *Computer World*.

[62] Roesch, M. (1999). Snort - lightweight intrusion detection for networks. *In Proceedings of the 13th USENIX conference on System administration*, pages 229–238, 1039864. USENIX Association.

[63] Roussev, V. (2009). Hashing and data fingerprinting in digital forensics. *Security and Privacy, IEEE*, 7(2):49–55.

[64] Roussev, V. (2010). *Data Fingerprinting with Similarity Digests*, volume 337/2010, pages 207–226. Springer.

[65] Roussev, V. (2011). An evaluation of forensic similarity hashes. *Digital Investigation*, 8(SUPPL.):S34–S41.

[66] Schleimer, S., Wilkerson, D. S. et Aiken, A. (2003). Winnowing : local algorithms for document fingerprinting. *In Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, 872770. ACM.

[67] Smith, R., Goyal, N., Ormont, J., Sankaralingam, K. et Estan, C. (2009). Evaluating GPUs for network packet signature matching. *In IEEE International Symposium on Performance Analysis of Systems and Software, 2009. ISPASS 2009.*, pages 175–184.

[68] Takaoka, T. (1986). An on-line pattern matching algorithm. *Information Processing Letters*, 22(6):329–330.

[69] Tan, L. et Sherwood, T. (2005). A high throughput string matching architecture for intrusion detection and prevention. *ACM SIGARCH Computer Architecture News*, 33(2):112–122.

[70] TCPDump (2012). libpcap. Tiré de `http://www.tcpdump.org/`, consulté le 13 août 2012.

[71] Tridgell, A. (2002). spamsum. Tiré de `http://www.samba.org/junkcode/#spamsum`, consulté le 7 avril 2012.

[72] Tumeo, A., Secchi, S. et Villa, O. (2011). Experiences with string matching on the fermi architecture. *In* Berekovic, M., Fornaciari, W., Brinkschulte, U. et Silvano, C., éditeurs : *Architecture of Computing Systems - ARCS 2011*, volume 6566 de *Lecture Notes in Computer Science*, pages 26–37. Springer Berlin / Heidelberg.

[73] Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E. P. et Ioannidis, S. (2008). Gnort : High performance network intrusion detection using graphics processors. *In Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 116–134, 1433016. Springer-Verlag.

[74] Vasiliadis, G. et Ioannidis, S. (2010). *GrAVity : A Massively Parallel Antivirus Engine - Recent Advances in Intrusion Detection*, volume 6307 de *Lecture Notes in Computer Science*, pages 79–96. Springer Berlin / Heidelberg.

[75] Wang, L., Chen, S., Tang, Y. et Su, J. (2011). Gregex : GPU based high speed regular expression matching engine. *In Proceedings of the 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 366–370, 2059033. IEEE Computer Society.

[76] Williams, R. (2010). Intel's core i7-980x extreme edition – ready for sick scores ? *Techgage*, (mars 2010):8.

[77] Xinyan, Z. et Sahni, S. (2011). Multipattern string matching on a GPU. *In IEEE Symposium on Computers and Communications (ISCC), 2011*, pages 277–282.

[78] Yoshihama, S., Mishina, T. et Matsumoto, T. (2010). Web-based data leakage prevention. *In IWSEC*.

[79] Yun, Q., Yang, Y. E. et Prasanna, V. K. (2012). Large-scale multi-flow regular expression matching on fpga. *In High Performance Switching and Routing (HPSR), 2012 IEEE 13th International Conference on*, pages 70–75.

[80] Zheng, Q. (2010). An improved multiple patterns matching algorithm for intrusion detection. *In IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS), 2010*, volume 2, pages 124–127.

ANNEXE A

FONCTIONS GPU

A.1 Calcul des signatures

La fonction calculant les signatures lit en entrée les données et les paramètres tels que le nombre de blocs ou les variables de calcul, et remplit le tableau de signatures et leurs positions. Son fonctionnement est décrit dans la section 3.3.1.

Elle prend en charge les recouvrements entre les blocs pour ne pas oublier de signatures dans le calcul du maximum.

Les fonctions appelées pour la mise à jour de la signatures par *rolling hash* et de calcul du maximum sont décrites dans les sous-sections A.2 à A.4.

A.1.1 Définitions préalables

La fonction gère les sauvegardes des multiples signatures dans un tableau et dans des registres. De même on peut activer ou non le filtrage des octets d'entrées et changer le filtre.

```
#define HASHINARRAY
//#define    FILTER_ON
#ifdef FILTER_ON
    #define      DISCARD_CHAIN(a,b)  a == 0 || a == 9 || a == 10
                                        || a == 13 || a == 32 || a == b
    // #define  DISCARD_CHAIN(a,b)  false
    // #define  DISCARD_CHAIN(a,b)  (a<32)||(a==b)
#endif
```

A.1.2 Prototype

```
__global__ void hashPacketKernel(const ulonglong2 *data,
                                 unsigned int nbPacket, uint2 params,
                                 unsigned long long *hashs,
                                 unsigned int *positions,
                                 unsigned int sizePacketDividedBy16)
{
```

A.1.3 Déclarations et Initialisations

```
1  /// This function computes signatures from data
2
3      unsigned int packet_id   = blockIdx.x * blockDim.x + threadIdx.x;
4
5      if(packet_id < nbPacket) {
6
7          int packetoffset = packet_id * sizePacketDividedBy16;
8  #ifdef HASHINARRAY
9          ulonglong2 hp[NBHASH];
10         #pragma unroll
11         for (int i = 0; i< NBHASH; i++) hp[i] = make_ulonglong2(0,0);
12 #else
13         unsigned long long h_0(0), h_1(0), h_2(0), h_3(0), pos_0(0), pos_1
               (0), pos_2(0), pos_3(0);
14 #endif
15         unsigned long long pos(0);
16         uint2 t = make_uint2(0,0);  // complete signature (h_up and h_down)
17         unsigned char index_last(0);// dynamic index for saving registers
18         unsigned char last[16] =    // backup of registers
19                         {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
20         unsigned char c_in, c_out;
```

A.1.4 Prise en compte des recouvrements

```
if(packet_id > 0) {

    // Case : not beginning of a block, overlap with previous block
    {   ulonglong2 data_in = data[packetoffset-1]; // longest read
        #pragma unroll
        for(int i=0; i<8; i++) {
            get_in( data_in.x, c_in, last, index_last, i );
            update_hash( c_in, 0, t, params );
        }
        #pragma unroll
        for(int i=0; i<8; i++) {
            get_in( data_in.y, c_in, last, index_last, i );
            update_hash( c_in, 0, t, params );
        }
    }

    // 16 B at beginning of the current block.
    {   ulonglong2 data_in = data[packetoffset]; // longest read
        #pragma unroll
        for(int i=0; i<8; i++) {
            get_in_out(data_in.x, c_in,c_out, last, index_last, i);
            update_hash( c_in, c_out, t, params );
            pos = ((unsigned long long )&data[packetoffset])-16+i;
#ifdef HASHINARRAY
            select_max(t, pos, hp);
#else
            select_max(t, pos, h_0,h_1 h_2,h_3,pos_0,pos_1,pos_2,pos_3);
#endif
        }
        #pragma unroll
        for(int i=0; i<8; i++) {
            get_in_out( data_in.y, c_in, c_out, last, index_last, i );
            update_hash( c_in, c_out, t, params );
            pos = ((unsigned long long )&data[packetoffset])-8+i;
#ifdef HASHINARRAY
            select_max(t, pos, hp);
#else
            select_max(t, pos, h_0,h_1,h_2,h_3,pos_0,pos_1,pos_2,pos_3);
#endif
        }
    }
```

```
42 } else {
43     // Case : beginning of packet, no overlap
44     ulonglong2 data_in = data[packetoffset]; // longest read
45     #pragma unroll
46     for(int i=0; i<8; i++) {
47         get_in( data_in.x, c_in, last, index_last, i );
48         update_hash( c_in, 0, t, params );
49     }
50     #pragma unroll
51     for(int i=0; i<8; i++) {
52         get_in( data_in.y, c_in, last, index_last, i );
53         update_hash( c_in, 0, t, params );
54     }
55 }
```

A.1.5 Milieu du bloc de données

```
1  for(int j = 1; j<sizePacketDividedBy16; j++) {
2      ulonglong2 data_in = data[packetoffset+j];
3      #pragma unroll
4      for(int i=0; i<8; i++) {
5          get_in_out( data_in.x, c_in, c_out, last, index_last, i );
6          update_hash( c_in, c_out, t, params );
7          pos = ((unsigned long long )&data[packetoffset+j])-16+i;
8  #ifdef HASHINARRAY
9          select_max(t, pos, hp);
10 #else
11         select_max(t, pos, h_0, h_1, h_2, h_3, pos_0, pos_1, pos_2, pos_3);
12 #endif
13     }
14     #pragma unroll
15     for(int i=0; i<8; i++) {
16         get_in_out( data_in.y, c_in, c_out, last, index_last, i );
17         update_hash( c_in, c_out, t, params );
18         pos = ((unsigned long long )&data[packetoffset+j])-8+i;
19 #ifdef HASHINARRAY
20         select_max(t, pos, hp);
21 #else
22         select_max(t, pos, h_0, h_1, h_2, h_3, pos_0, pos_1, pos_2, pos_3);
23 #endif
24     }
25 }
```

A.1.6 Sauvegarde des maxima et de leurs positions

```
#ifdef HASHINARRAY
        #pragma unroll
        for(int h=0; h<NBHASH; h++) {
            hashs[packet_id * NBHASH + h] = hp[h].x;
            positions[packet_id * NBHASH + h] =
                    (unsigned int)(hp[h].y - (unsigned long long)data);
        }
#else
        hashs[packet_id * NBHASH + 0] = h_0;
        hashs[packet_id * NBHASH + 1] = h_1;
        hashs[packet_id * NBHASH + 2] = h_2;
        hashs[packet_id * NBHASH + 3] = h_3;
        positions[packet_id * NBHASH + 0] =
                    (unsigned int)( pos_0  - (unsigned long long)data);
        positions[packet_id * NBHASH + 1] =
                    (unsigned int)( pos_1  - (unsigned long long)data);
        positions[packet_id * NBHASH + 2] =
                    (unsigned int)( pos_2  - (unsigned long long)data);
        positions[packet_id * NBHASH + 3] =
                    (unsigned int)( pos_3  - (unsigned long long)data);
#endif
    }
}
```

A.2 Lecture et sauvegarde des octets entrants

Ces fonctions préparent les octets entrant et sortant de la fenêtre de calcul.

A.2.1 Cas complet

```
__inline__ __device__
void get_in_out( unsigned long long data_in,
                 unsigned char &c_in, unsigned char &c_out,
                 unsigned char *last, unsigned char &index_last, int i )
{
    c_in  = (unsigned char)(data_in >> (8*i));
#ifdef FILTER_ON
    if (DISCARD_CHAIN(c_in,last[index_last])) continue;
    index_last = (index_last + 1)%16;
    c_out = last[index_last];
    last[index_last] = c_in;
#else
    c_out = last[i+8];
    last[i+8] = c_in;
#endif

}
```

A.2.2 Uniquement un octet entrant

```
__inline__ __device__
void get_in( unsigned long long data_in,
             unsigned char &c_in,
             unsigned char *last, unsigned char  &index_last, int i )
{
    c_in = (unsigned char)(data_in >> (8*i));
#ifdef FILTER_ON
    if (DISCARD_CHAIN(c_in,last[index_last])) continue;
    index_last = (index_last + 1)%16;
    last[index_last] = c_in;
#else
    last[i] = c_in;
#endif
}
```

A.3 Mise à jour de la signature

Cette fonction implémente le *rolling hash* pour mettre à jour le *hash* à chaque déplacement de la fenêtre de calcul.

```
__inline__ __device__
void update_hash( unsigned char c_in,
                  unsigned char c_out,
                  uint2 &t, uint2 params )
{
    t = make_uint2(
       ((t.x << 4) | (t.y >> 28)) ^ (c_in * params.x) ^ (c_out * params.x),
       ((t.y << 4) | (t.x >> 28)) ^ (c_in * params.y) ^ (c_out * params.y)
    );
}
```

A.4 Mise à jour du maximum

Ces fonctions gèrent le calcul du maximum et de la position de celui-ci à chaque déplacement de la fenêtre de calcul, afin que le résultat soit obtenu dès que le déplacement est terminé sans calcul supplémentaire.

A.4.1 Avec un tableau de maxima

```
#ifdef HASHINARRAY
__inline__ __device__
void select_max(uint2 t, unsigned long long pos, ulonglong2 *hp)
{
    unsigned long long sigTemp = t.x;
    sigTemp = (sigTemp << 32) | t.y;
    hp[t.x >> NBBITS].x = max(hp[t.x >> NBBITS].x, sigTemp);
    if(hp[t.x >> NBBITS].x == sigTemp) hp[t.x >> NBBITS].y = pos;
}
#endif
```

A.4.2 Avec des registres

```
#ifndef HASHINARRAY
__inline__ __device__
void select_max(uint2 t, unsigned long long pos,
                unsigned long long &h_0, unsigned long long &h_1,
                unsigned long long &h_2, unsigned long long &h_3,
                unsigned long long &pos_0, unsigned long long &pos_1,
                unsigned long long &pos_2, unsigned long long &pos_3)
{
    unsigned long long sigTemp = t.x;
    sigTemp = (sigTemp << 32) | t.y;
    switch(t.x >> NBBITS) {
        case 0:
            h_0 = max(h_0, sigTemp);
            if(h_0 == sigTemp) pos_0 = pos;
            return;
        case 1:
            h_1 = max(h_1, sigTemp);
            if(h_1 == sigTemp) pos_1 = pos;
            return;
        case 2:
            h_2 = max(h_2, sigTemp);
            if(h_2 == sigTemp) pos_2 = pos;
            return;
        case 3:
            h_3 = max(h_3, sigTemp);
            if(h_3 == sigTemp) pos_3 = pos;
            return;
    }
}
#endif
```

A.5 Recherche de signatures

Cette fonction cherche une signature dans un emplacement de la ligne adéquate de la *hash table*. Son fonctionnement est décrit à la section 3.3.2. On lance un nombre d'instance supérieur ou égal au nombre de signatures à chercher multiplié par le nombre de colonnes de la *hash table*. Dans un bloc, `threadIdx.x` identifie la colonne à analyser. L'ensemble des autres indices (sur `x` et `y`) identifie la signature à analyser

```
1  __global__
2  void searchHashKernel(const unsigned long long *hashs, unsigned int nbHash,
3                        const unsigned long long *db, const unsigned int *rf,
4                        unsigned int nbLSB, unsigned int nbColonnes,
5                        unsigned int *found, unsigned int *flagFound)
6  {
7      unsigned int hash = blockIdx.y * gridDim.x * blockDim.y
8                                + blockIdx.x * blockDim.y
9                                + threadIdx.y;
10     unsigned int colonne = threadIdx.x;
11     if(hash < nbHash) {
12         unsigned long long hashToSearch = hashs[hash];
13         unsigned int ligne = hashToSearch & ((1 << DB_LSB) - 1);
14         if(hashToSearch > 0) {
15             if(db[DB_COLUMNS * ligne + colonne] == hashToSearch) {
16                 found[hash] = rf[hash];
17                 *flagFound  = 1;
18             }
19         }
20     }
21 }
```

ANNEXE B

FONCTIONS CPU

B.1 Analyse d'un buffer

Lorsqu'un *buffer* est rempli, on appelle la fonction d'analyse. Celle-ci se charge d'envoyer les données vers le GPU et de lancer les *kernels* de calcul et de recherche des signatures.

B.1.1 Prototype et initialisation

Les *buffers* sur le GPU sont déjà alloués, la fonction qui s'en charge n'est pas décrite ici. On indique simplement l'identifiant des *buffers* que l'on souhaite utiliser.

La classe `Buffer` n'est pas décrite ici. Elle contient notamment les données (`mem`) et leur taille (`size`).

```
1  void HashTable::search(Buffer *sourceBuffer, int bufferIndex) {
2
3      if( sourceBuffer->size <= SIZEPACKET ) {
4          // Pas d'analyse si le buffer contient moins qu'un paquet
5          nHash[bufferIndex] = 0;
6          return;
7      }
8
9      // Envoi des donnees vers la memoire du GPU deviceBuffer[bufferIndex]
10     // L'envoi est asynchrone puisque le DMA s'en charge
11     cudaMemcpyAsync( deviceBuffer[bufferIndex],
12                      sourceBuffer->mem,
13                      sourceBuffer->size,
14                      cudaMemcpyHostToDevice,
15                      stream[bufferIndex]
16                    );
17
18     // On verifie que le lancement s'est bien deroule
19     checkCUDAError("HashTable::search:cudaMemcpyAsync deviceBuffer", true);
```

B.1.2 Calcul des signatures

```
// Configuration de la grille de lancement
unsigned int nbPacket = sourceBuffer->size / SIZEPACKET;
unsigned int nbHashs = nbPacket * NBHASH;

// Lancement
hashPacketKernel
    <<< 14, (int)ceil((float)nbPacket/1400.0f), 0, stream[bufferIndex] >>>
    (   deviceBuffer[bufferIndex], nbPacket, make_uint2(params_x, params_y)
        ,
        deviceHashes[bufferIndex], devicePositions[bufferIndex]
    );

// On verifie que le lancement s'est bien deroule
checkCUDAError("HashTable::search:hashPacketKernel launch", true);
```

B.1.3 Recherche des signatures

```
// On remet le drapeau a 0
*(deviceFlag[bufferIndex]) = 0;

// Configuration de la grille de lancement
dim3 nbSearchBlock(nbColonnes, ceil((float)nbHashs/512.0f), 1);

// Lancement
searchHashKernel
    <<< nbSearchBlock, 512, 0, stream[bufferIndex] >>>
    (   deviceHashes[bufferIndex], nbHashs, dbPtr,
        devicePositions[bufferIndex], nbLSB, nbColonnes,
        deviceFound[bufferIndex], deviceFlag[bufferIndex]
    );

// On verifie que le lancement s'est bien deroule
checkCUDAError("HashTable::search:searchHashKernel launch", true);

// Attente que le stream soit termine
cudaStreamSynchronize( stream[bufferIndex] );
checkCUDAError("HashTable::analyzeResults:cudaStreamSync", true);
```

B.1.4 Attente et analyse des résultats

```
1     // On lit le drapeau
2     if(*(deviceFlag[bufferIndex]) > 0) {
3
4         // On ne recupere les resultats que si necessaire
5         cudaMemcpyAsync( hostFound[bufferIndex],
6                          deviceFound[bufferIndex],
7                          nbHashs*sizeof(unsigned int),
8                          cudaMemcpyDeviceToHost,
9                          stream[bufferIndex]
10                       );
11        checkCUDAError("HashTable::analyzeResults:cudaMemcpy", true);
12
13        // Attente de la fin du transfert
14        cudaStreamSynchronize(stream[bufferIndex]);
15        checkCUDAError("HashTable::analyzeResults:cudaStreamS", true);
16
17        // On remet a zero les buffers de resultats
18        // Fonction non decrite ici
19        xferBufferReset(bufferIndex);
20
21        // Analyse des resultats
22        for(int i=0; i<nbHashs; i++) {
23            if(hostFound[bufferIndex][i] != 0) {
24
25                // En cas de correspondance, on affiche une alerte
26                LOG_INFO("Hit @ offset = "
27                         << hostFound[bufferIndex][i]
28                         << endl);
29                // On retrouve le paquet IP incrimine
30                IPPacket *hit = sourceBuffer->find_offset(
31                                                  hostFound[bufferIndex][i]);
32                // Et on bloque la communication correspondante
33                if (hit != NULL) blocker.add( hit );
34            }
35        }
36    // Si le flag est a zero, il n'y a rien a faire
37    } else {
38        // remise a zero des buffers d'analyse (fonction non presentee ici)
39        xferBufferReset(bufferIndex);
40    }
41 }
```