Universidade
Federal do
Pampa

Autor: Leandro Augusto Stachlewski Gomes

RECRIPTOGRAFIA ASSISTIDA POR GPU EM SERVIDORES DE DISTRIBUIÇÃO DE VÍDEO

Trabalho de Conclusão de Curso II

BAGÉ
2013

Leandro Augusto Stachlewski Gomes

Recriptografia Assistida por GPU em Servidores de Distribuição de Vídeo

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Computação.

Orientador: Dr. Leonardo Bidese de Pinho

Bagé
2013

LEANDRO AUGUSTO STACHLEWSKI GOMES

RECRIPTOGRAFIA ASSISTIDA POR GPU EM SERVIDORES DE DISTRIBUIÇÃO DE VÍDEO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Computação.

Trabalho de Conclusão de Curso defendido e aprovado em: 19 de Outubro de 2013.
Banca examinadora:

____

Prof. Dr. Leonardo Bidese de Pinho
Orientador
Engenharia de Computação/Campus Bagé – UNIPAMPA

____

Profa. Dra. Ana Paula Lüdtke Ferreira
Engenharia de Computação/Campus Bagé – UNIPAMPA

____

Prof. MSc. Érico Marcelo Hoff do Amaral
Engenharia de Computação/Campus Bagé – UNIPAMPA

Dedico esse trabalho à minha família, e em especial a Neusa, Fábio e Náthalie fontes de apoio emocional e financeiro durante toda a minha vida, aos meus amigos pelo companheirismo, amizade e incentivo, e a todas as pessoas que me ajudaram de alguma forma, direta ou indiretamente. Ao professor Leonardo pela incansável ajuda e orientação, e a todos os doutores, mestres, professores e funcionários da UNIPAMPA que tive oportunidade de conviver por esse período.

RESUMO

A distribuição segura de vídeos para grandes audiências envolve alta capacidade de processamento paralelo em módulos de recriptografia capazes de individualizar os fluxos de vídeo aos clientes, evitando a necessidade de compartilhamento de chaves. A natureza paralela do problema faz dele candidato potencial para implementações que explorem o *hardware* paralelo disponível nos sistemas computacionais atuais. Com a disseminação das GPUs, surge a necessidade de avaliar empiricamente não apenas o desempenho neste *hardware*, mas também outros requisitos fundamentais como consumo energético e custo total de implementação. Neste trabalho são comparadas alternativas de implementação do módulo de recriptografia com CUDA, Pthreads e OpenMP e coletados resultados empíricos para diferentes quantidades de clientes concorrentes (fornecendo vazão de criptografia de 3,2 Mbps com uma GPU Fermi e 1,8 Mbps com um *multicore* em um cenário com 1024 clientes), sugerindo que o uso das GPUs pode diminuir o custo total de implementação. Além disso, o presente trabalho propõe e valida, com precisão satisfatória (divergindo em média de 9% e 14% para determinadas GPUs da família GT200 e Fermi respectivamente), um modelo matemático para estimar genericamente o desempenho fornecido por uma GPU para o módulo de criptografia implementado. Para completar a sustentação da viabilidade da implementação do módulo proposto, este trabalho se propõe a fazer o levantamento do consumo energético de cada configuração experimentada através de um simulador de GPU, no qual os resultados apontam para um consumo de energia, aproximadamente, em média, de 60 W por *kernel*. Uma vez conhecidas as três principais métricas (desempenho, consumo energético e preço) será possível determinar a solução de melhor custo-benefício para o módulo de recriptografia.

Palavras-chave: criptografia. GPU. *Multicore*. Vídeo sob demanda.

ABSTRACT

Proxy servers of video distribution systems must provide efficient memory management and also have to adopt protection mechanisms so that only allowed clients would have access to restricted video content. The intrinsic parallel nature of this problem plays a role on implementations that exploit parallel hardware available in off-the-shelf computing systems. With the spread of multicore GPUs, not only there is a need for empirical performance evaluation but also for other fundamental requirements such as energy consumption and total cost of implementation. In this work implementation alternatives of the recryptography module are compared, based on CUDA, Pthreads and OpenMP, using collected empirical results for different amounts of concurrent customers (given the cryptography throughput of 3,2 Mbps for a Fermi GPU and 1,8 Mbps for a multicore in a 1024 clients scenario), suggesting that the use of GPUs can reduce the total implementation cost of the module. Moreover, the present work proposes and validates, with satisfactory accuracy (ie with 9% and 14% diverging average for a given GPU of GT200 and Fermi families respectively), an adaptation of a mathematical model to forecast performance provided by different GPU architectures for the cryptography module implemented, which is cross-validated with empirical results collected for different architectures. To complete the feasibility analysis of a real implementation of the proposed module, this work intends to estimate the energy consumption of each experienced configuration by a GPU simulator where the results points to an average energy consumption about 60 W per kernel. Once known the main three metric (performance, energy consumption and price) it will be possible to determine the solution of better cost-benefit for the recryptography module.

Keywords: cryptography. GPU. Multicore. Video on demand.

LISTA DE FIGURAS

Figura 1 - Caminho percorrido pela matriz de estados 9
Figura 2 - Tabela de referência para substituição 10
Figura 3 - Funcionamento do *ShiftRows* 10
Figura 4 - Funcionamento do *MixColumns* 11
Figura 5 – Diferença arquitetural entre CPU *multicore* e GPU 12
Figura 6 – Modelo de *hardware* de uma GPU 13
Figura 7 – Modelo de memória de uma GPU 14
Figura 8 – Arquitetura da família Fermi 16
Figura 9 – Esquemático do NVCC 20
Figura 10 - Abstração do modelo CUDA na GPU 21
Figura 11 – Distribuição de vídeos usando proxies com recriptografia 24
Figura 12 – Fluxograma do funcionamento do módulo de recriptografia 28
Figura 13 – Fluxograma de funcionamento da versão *coarse-grained* 30
Figura 14 – Fluxograma de funcionamento da versão *fine-grained* 31
Figura 15 – Fluxograma de funcionamento da versão *mix-grained* 32
Figura 16 – Fluxograma de funcionamento da abordagem A da *mix-grained* adaptativa 33
Figura 17 – Fluxograma de funcionamento da abordagem B da *mix-grained* adaptativa 34
Figura 18 – Fluxograma de funcionamento da abordagem C da *mix-grained* adaptativa 35
Figura 19 – Comparativo das melhores versões (servidor) 39
Figura 20 – Comparativo das melhores versões (*desktop*) 39
Figura 21 – Efeito da relação entre *warps* e núcleos da GPU 40
Figura 22 - Consumo Energético GTX 480 (MG_C_NCP=8) 43
Figura 23 - Consumo Energético GTX 480 (MG_C_NCP=512) 43
Figura 24 – Resultados iniciais (C2075 sem correção do fator multiplicativo) 47
Figura 25 – Resultados iniciais (C2075 com correção do fator multiplicativo) 48
Figura 26 - Comparação entre o modelo matemático e simulação 50

LISTA DE TABELAS

Tabela 1 – Principais diferenças entre as famílias de GPUs....................................17
Tabela 2 - Vazão demandada para streaming com diferentes padrões de compressão e perfis de vídeos..37
Tabela 3 – Especificações das GPUs e mapeamento para o modelo.......................46

LISTA DE SIGLAS

AES – *Advanced Encryption Standard*

API – *Application Programming Interface*

CD – *Compact Disc*

CE – Consumo Energético

CB – *Communication Bound*

CPU – *Central Processing Unit*

CTR – Counter

CUDA – *Compute Unified Device Architecture*

DRM – *Digital Rights Management*

DVD – *Digital Versatile Disc*

ECB – *Electronic Code Book*

ECC – *Error-correcting Code*

FPS – *Frames Per Second*

GPGPU – *General Purpose Graphics Processing Unit*

GPU – *Graphics Processing Unit*

NCP – Número de clientes em paralelo

NCT – Número de *threads* cooperativas

NVCC – *Nvidia C Compiler*

PB – *Processing Bound*

PE – *Processing Element*

QoS – *Quality of Service*

SIMD – *Single Instruction Multiple Data*

SM – *Simultaneous Multiprocessor*

SSL – *Secure Sockets Layer*

TCC – Trabalho de Conclusão de Curso

TI – Tecnologia da Informação

TLS – *Transport Layer Security*

SUMÁRIO

1 INTRODUÇÃO 1
1.1 Principais contribuições..3
1.2 Organização do trabalho 4
2 CONHECIMENTOS BÁSICOS 5
2.1 Distribuição de vídeo sob demanda 6
2.1.1 AES (Advanced Encryption Standard) 8
2.2 GPU..11
2.3 APIs de programação paralela 17
2.3.1 APIs para CPUs multicores 19
2.3.2 API GPGPU (CUDA)...20
3 TRABALHOS RELACIONADOS 22
4 RECRIPTOGRAFIA DE VÍDEO ASSISTIDA POR GPU...............................24
4.1 Coarse-grained 29
4.2 Fine-grained...30
4.3 Mix-grained 31
4.4 Mix-grained adaptativas...32
4.5 Métricas de avaliação...35
4.5.1 Análise de desempenho 35
4.5.2 Análise do consumo energético 41
4.6 Restrições dos experimentos 43
5 MODELO MATEMÁTICO (ANALÍTICO)..45
6 CONSIDERAÇÕES FINAIS 51
REFERÊNCIAS...53

1 INTRODUÇÃO

C*omputação verde* (*green computing ou green IT*) é um dos tópicos que cada vez mais vem atraindo a atenção dos pesquisadores, com destaque para a computação com eficiência energética (*energy-efficient computing*) (MURUGESAN, 2008). A computação verde refere-se às ações de responsabilidade ambiental como: a redução de consumo energético, desenvolvimento de sistemas e componentes de baixo custo, reciclagem, redução de resíduos, entre outros. Muitas vezes apresentar eficiência energética não é suficiente, sendo necessário balancear a questão do consumo energético com o desempenho e o preço, o que se convencionou denominar computação de alta eficiência (*high efficiency computing*). Esta tem sido foco crescente de estudos da comunidade envolvida em pesquisa na área de arquiteturas de sistemas computacionais, de modo que diferentes aplicações vêm sendo revisitadas com o intuito de propor, desenvolver e avaliar soluções eficientes com base nas novas tecnologias que estão à disposição (ACCELERATORS…, 2009).

Sob esta perspectiva, a amplitude de aplicabilidade de uma tecnologia em particular tem sido base para diversos estudos: as Unidades Gráficas de Processamento (*graphics processing units* - GPUs). As GPUs, presentes nas placas de vídeo, vêm chamando a atenção de vários pesquisadores como alternativa para aumentar o desempenho de aplicações e possibilitar alto desempenho com baixo investimento financeiro em recursos de *hardware* (ACCELERATORS…, 2009). O espaço de aplicação das GPUs tem se ampliado, com o passar do tempo, em razão do surgimento de APIs (*application programming interfaces*) que facilitam a programação para esses componentes (MANAVSKI, 2007). Resumidamente, a ideia básica é aproveitar a significativa e crescente capacidade de processamento (NVIDIA, 2011), muitas vezes ociosa, oferecida por estes dispositivos de *hardware*, originalmente introduzidos nos sistemas computacionais para o processamento gráfico, como elemento de co-processamento para execução eficiente de aplicações de propósito geral que anteriormente ficavam restritas à execução em um ou mais núcleos disponíveis na unidade central de processamento (*central processing unit* - CPU) do sistema.

Diferentes pesquisas têm surgido neste contexto, denominado de GPU para Propósito Geral (*general purpose GPUs* – GPGPUs), cobrindo diferentes aplicações.

Em geral, o principal foco destas pesquisas está na questão do desempenho, embasadas em métricas convencionais como tempo de execução ou específicas da aplicação em questão.

Paralelamente ao que foi exposto, com a popularização da distribuição de vídeo em redes de larga escala como a Internet (GILL, 2007), cresce a demanda por sistemas escaláveis capazes de atender grandes quantidades de clientes concorrentemente. Para tanto, soluções baseadas em servidores intermediários (*proxies*) têm sido propostas e avaliadas na literatura tal que estas se apresentam como as de melhor custo-benefício. Atrelada à problemática da distribuição escalável dos conteúdos, está outra questão de pesquisa fundamental: a proteção dos direitos autorais (*digital rights management* – DRM) dos vídeos. O problema do DRM é normalmente resolvido por meio de soluções que envolvem módulos de criptografia e que, em geral, demandam capacidade computacional proporcional à quantidade de usuários sendo atendidos pelo *proxy*.

Neste cenário, o trabalho Gomes (2012c), o qual precedeu este trabalho, se concentrou no estudo do problema da recriptografia (*reencryption*) – necessidade de decifrar e cifrar novamente uma sequência de dados – assistida por GPU em *proxies* de vídeo que, conforme o levantamento bibliográfico realizado sugere, ainda não foi abordado de forma experimental por outros pesquisadores sob o ponto de vista de métricas de desempenho convencionais para a aplicação alvo (tal como a vazão de vídeo recriptografado, que pode ser traduzida em taxa de atendimento/bloqueio de requisições concorrentes de usuários).

Dependendo do algoritmo adotado, a demanda por processamento na execução do módulo de criptografia causada por usuários concorrentes pode se tornar o principal limitante para o sistema, superando o gargalo principal que é normalmente definido pela capacidade de vazão da interface de rede. Um exemplo é o algoritmo de criptografia *Advanced Encryption Standard* (AES) (DAEMEN, 2002), foco deste estudo, o qual, além de ser um dos mais utilizados nessa área, apresenta forte potencial para exploração de paralelismo (MANAVSKI, 2007) e oferece uma implementação sequencial de referência, presente na biblioteca OpenSSL (2011).

Conforme é demonstrado adiante, atingir uma implementação custo-efetiva do módulo depende da solução de processamento adotada para o módulo de criptografia, a qual deve ser capaz de explorar os diferentes elementos de processamento (PEs) disponíveis (sejam estes elementos de processamento da

CPU ou da GPU), o que implica em usar APIs tradicionais de processamento paralelo (Ex.: Pthreads (BUTTENHOF, 1997) e OpenMP (2011) e/ou APIs para processamento de propósito geral em GPUs (Ex.: NVIDIA (2011a)), a luz das características da aplicação. Em trabalhos anteriores (GOMES, 2011;GOMES, 2012a; GOMES, 2012b; GOMES, 2012c) foi demonstrado que a exploração da GPU usando CUDA (*compute unified device architecture)*, em comparação com implementações em Pthreads e OpenMP, permite obter uma vazão de criptografia significativamente superior, viabilizando o atendimento de grandes quantidades de clientes concorrentes. Em particular, foi demonstrado em Gomes (2012b) que essa diferença de vazão é afetada pela relação existente entre a estratégia de paralelização do AES e a arquitetura da GPU, principalmente no que se refere à quantidade de elementos de processamento e ao mecanismo de escalonamento. Neste contexto, Gomes (2012c) ampliou o estudo de adaptações alternativas que repercutem em melhor ou pior aproveitamento da capacidade computacional disponível, ampliando o estado-da-arte com três principais contribuições:

1. Proposta de duas novas versões *mix-grained* adaptativas – políticas de paralelização do algoritmo AES adaptáveis a diferentes quantidades de *núcleos* ;
2. Implementação das novas políticas em CUDA;
3. Avaliação empírica das implementações em dois sistemas computacionais, incluindo a análise do mecanismo de escalonamento de *thread*s da GPU.

1.1 Principais contribuições

A partir deste contexto, foi identificada a possibilidade de contribuir com o estado-da-arte na área por meio da criação de um modelo matemático para descrever o comportamento da aplicação do módulo de recriptografia, de modo análogo ao modelo proposto em Barlas (2011), presente na revisão bibliográfica. Esta nova abordagem permite atingir contribuições ainda mais significativas para o trabalho, uma vez que com a criação de um modelo matemático é possível generalizar as análises. Ou seja, passa a ser possível prever o comportamento da aplicação em outros cenários (por exemplo, uma GPU diferente), sem a necessidade de aquisição de um novo componente de *hardware*, uma vez que os resultados obtidos fornecem uma estimativa de desempenho com divergência, em média, de

9,4% para GPUs da família GT200 e 14,1% para as da família Fermi.

Por outro lado, este trabalho também realiza a mensuração do consumo energético dos diversos cenários experimentados através de um simulador de GPU (GPGPU-Sim) (BAKHOD, 2009), no qual os resultados apontam para um consumo de energia por *kernel*, aproximadamente, em média, de 60 W. Além disso, foi percebido que tanto o desempenho quanto o consumo energético, dependem não só da arquitetura da GPU, mas também do fator NCP (número de clientes em paralelo), e quanto maior o valor desse fator (respeitando as características inerentes da GPU alvo), maior tende a ser a eficiência da implementação.

1.2 Organização do trabalho

No capítulo 2, são apresentados os principais conceitos que servem como base para o entendimento deste trabalho, como as métricas de avaliação do módulo de recriptografia, os princípios de programação paralela, as APIs e as características dos ambientes experimentais utilizados neste trabalho. Em seguida, no capítulo 3, são apresentados os principais trabalhos relacionados. O capítulo 4 apresenta a aplicação foco deste trabalho que é a recriptografia de vídeo assistida por GPU aplicada no contexto de *proxies*. O capítulo 5 trata da metodologia de desenvolvimento do trabalho, ou seja, o modelo matemático (analítico). E por último, mas não menos importante, o capítulo 5 traz as considerações finais.

2 CONHECIMENTOS BÁSICOS

A seguir serão apresentados os conhecimentos básicos para o melhor entendimento deste trabalho de conclusão de curso, seus objetivos, metodologias de desenvolvimento, resultados, discussão e considerações finais.

Cada vez mais as pessoas tem se preocupado com a questão do consumo energético (CE), uma vez que uma grande parcela dos recursos para geração de energia vem da queima de combustíveis fósseis, que além de estarem, a cada dia, mais escassos, são responsáveis pela poluição e pelo efeito estufa que aumentam os buracos na camada de ozônio do nosso planeta. Segundo Murugesan (2008), o total de energia elétrica consumida por servidores, computadores, monitores, equipamentos de comunicação de dados e sistemas de refrigeração para *data centers* está gradativamente aumentando e isso reflete na premissa de que cada PC em uso gere, direta ou indiretamente, cerca de uma tonelada de dióxido de carbono a cada ano. Tendo em vista essa problemática, um grande número de vendedores de TI (Tecnologia da Informação) e usuários tem se voltado à TI verde, ajudando a construção de uma economia e sociedade conscientes. Uma das vertentes da TI verde é a computação com eficiência energética (*energy-efficient computing*), que traz uma das métricas para avaliação do módulo de recriptografia (consumo de energia), a qual influencia diretamente a escolha da melhor solução (uso de GPU ou CPU com vários núcleos de processamento) para implementação do módulo, uma vez que o *proxy* com recriptografia precisaria ficar ligado 24 horas por dia a fim de atender a constante demanda dos clientes.

Mas além de soluções energeticamente eficientes, os usuários demandam computadores, *notebooks* e dispositivos portáteis com melhor desempenho e com menor investimento em recursos de *hardware* e *software*, o que acaba por contribuir com o avanço da tecnologia, uma vez que os desenvolvedores de TI irão ser induzidos a se esforçar mais para atender a demanda desses usuários. Esse balanço entre desempenho e preço pode ser expresso com o termo *high efficiency computing*, o qual também propõe o projeto, análise e implementação de algoritmos paralelos com escalabilidade, adaptabilidade e portabilidade em arquiteturas com vários elementos de processamento. Ambos os fatores também são elementares para determinar a melhor configuração do módulo de recriptografia, pois o desempenho é importante para manter a qualidade de serviço (QoS) no atendimento

a grande quantidade de clientes, e o custo total de implementação, que é função do consumo de energia e da aquisição dos componentes de *hardware* para cada *proxy* com recriptografia.

2.1 Distribuição de vídeo sob demanda

Atualmente, a popularização da tecnologia e multimídia faz os usuários, cada vez mais, utilizarem a Internet para compartilhamento e visualização de vídeos através de *sites* como o Youtube, o qual é um exemplo clássico de distribuição de vídeo pela rede (GILL, 2007). Por outro lado, os serviços de *pay-per-view* para compra de séries, filmes, jogos de futebol e demais esportes, fornecidos por provedores de conteúdo multimídia como Netflix, TerraTV e, mais recentemente, o Now da NetTV, são exemplos de distribuição de vídeo sob demanda. A distribuição de vídeos sob demanda requer soluções com sistemas escaláveis capazes de atender grandes quantidades de clientes concorrentemente sem afetar a qualidade de serviço e, ao mesmo tempo, não violar os direitos autorais desses vídeos.

Para o funcionamento de um sistema de distribuição de vídeo sob demanda pela Internet, é necessário o uso de um ou mais servidores principais. Neste trabalho, o termo servidor é usado para representar um computador que realiza o controle do acesso de conteúdos específicos, requisitado por clientes, de uma determinada rede à Internet, podendo ser classificado de acordo com o tipo de conteúdo que fornece, como é o caso dos servidores de vídeo. O servidor também é responsável por armazenar o conteúdo a ser transferido aos clientes. Uma vez que o servidor é a principal fonte de conteúdo, os inúmeros clientes que demandam acesso a arquivos podem gerar uma sobrecarga no servidor. Para solucionar esse problema, várias soluções baseadas em *proxies* são propostas na literatura por pesquisadores na área de redes (ALVES, 2009).

Os *proxies* funcionam como servidores intermediários, responsáveis por manter uma parcela do conteúdo do servidor principal, tendendo a diminuir o acesso dos clientes ao servidor e assim diminuir a chance de sobrecarga dele. As soluções com proxies são as de melhor relação custo-benefício (GRANADO, 2010). Os *proxies* complementam a capacidade do servidor principal, onde os dados são armazenados de forma primária e não volátil. Eles também podem diminuir a latência de início de exibição (em sistemas de distribuição de vídeo em geral, mas

especialmente no caso de vídeo sob demanda) e também por reduzirem a largura de banda necessária no enlace até o servidor, devido a, geralmente, ele estar situado mais próximo ao cliente. Os *proxies* de vídeo sob demanda, por exemplo, adotam soluções especializadas de *software* responsáveis pelo gerenciamento do conteúdo que deve ser mantido em *cache*, prevendo um novo acesso aos trechos de vídeo em função de novas requisições de vídeo.

O uso de *proxies* auxilia a manter um nível de qualidade de serviço aceitável. A qualidade de serviço é um requisito da(s) aplicação(ões) para a qual exige-se que determinados parâmetros (atrasos, vazão, perdas, entre outros) estejam dentro dos limites bem definidos (valor mínimo, valor máximo). Do ponto de vista dos clientes, normalmente tem-se que a qualidade de uma aplicação pode ser variável e, repentinamente, pode ser alterada ou ajustada (para melhor ou pior qualidade). Por exemplo, pode-se assistir um vídeo com uma qualidade de 32 fps (*frames per second*) ou 4 fps e isto depende da qualidade de vídeo esperada pelo cliente. Com o avanço da tecnologia em termos de processamento de vídeo, os clientes solicitam qualidades de vídeo cada vez mais próximas da realidade, o qual se reflete em maior quantidade de informação (aumento no tamanho do vídeo) e taxa de exibição (SANTANA, 2006).

Além da QoS, um *proxy* de distribuição de vídeo sob demanda precisa fornecer mecanismos de proteção dos direitos autorais. Segundo Subramanya (2006), o gerenciamento dos direitos autorais (em geral) se refere a um conjunto de normas, técnicas e ferramentas que norteiam o uso adequado de conteúdo digital. A DRM tem um papel importante em muitos processos onde são envolvidos fluxo de conteúdo, como é o caso da distribuição de vídeo sob demanda. As principais funcionalidades de um sistema DRM são o empacotamento do conteúdo original para fácil distribuição e localização, proteção de conteúdo para uma transmissão segura, entrega de conteúdo *offline* em CDs e DVDs e distribuição de conteúdo sob demanda através de redes *peer-to-peer* (ALVES, 2009).

Uma maneira de proteger o conteúdo dos dados que trafegam na rede é utilizar a criptografia, como foi feito nesse trabalho. Criptografia (do latim *kryptos* + *grafos*) significa escrita desconhecida, é uma técnica que surgiu da necessidade de manter dados secretos para quem não se deseje revelar os seu conteúdo, como nas cartas cifradas em períodos de guerra ao longo da história da humanidade. Com o passar do tempo simples substituições de letras nas mensagens foram ganhando

mais segurança à medida que substituições maiores se tornaram possíveis, principalmente com o surgimento dos computadores. Conforme mencionado nas seções anteriores, para solucionar o problema do DRM em relação ao módulo de recriptografia de vídeo, foi adotado o algoritmo de criptografia AES, o qual será explicado na próxima subseção.

2.1.1 AES (*Advanced Encryption Standard*)

Como mencionado no Capítulo 1, o AES está presente na OpenSSL que é uma biblioteca de criptografia de código aberto que implementa os protocolos SSL (*Secure Socket Layer*) e o seu sucessor TLS (*Transport Layer Security*), comumente utilizado para garantir segurança no envio e recebimento de notas fiscais eletrônicas, além de ser a mais utilizada e que possui implementação disponível e gratuita desses protocolos (OPENSSL, 2011).

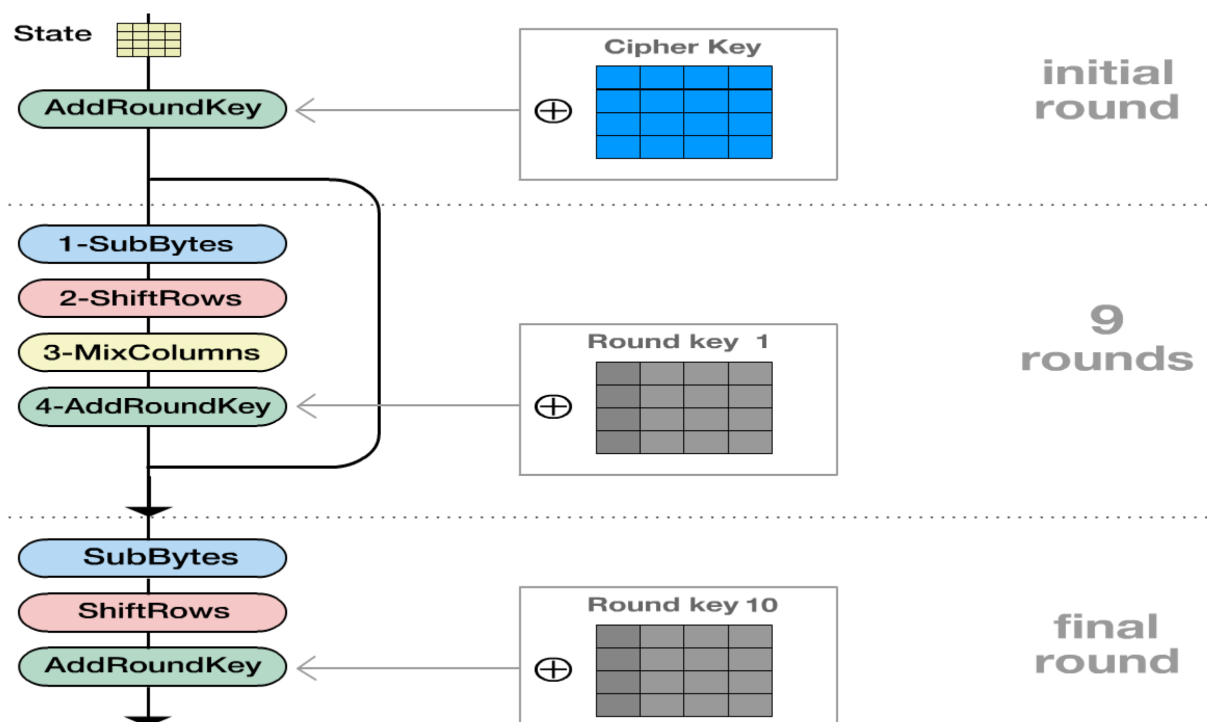
O AES surgiu através de um concurso lançado pelo NIST (*National Institute of Standards in Technology*) no ano de 1997 onde foi anunciado um plano para definir um algoritmo de chave simétrica para proteger informações do governo federal e tornar-se o novo padrão para criptografia que depois de três anos e meio após o início do concurso, passando por rigorosos critérios de avaliação (como segurança, desempenho e pouco consumo de memória), o NIST chegou à escolha do vencedor: Rijndael. O nome é uma fusão nominal de Vincent Rijmen e Joan Daemen, os dois belgas criadores do algoritmo.

O AES é um algoritmo de criptografia de bloco simétrico (mesma chave de criptografia e decriptografia) e tem um tamanho de bloco de entrada fixo (128 *bits* ou 16 *bytes*) e uma chave com tamanho de 128, 192 ou 256 *bits.* Como ele possui pouco poder de processamento e utiliza de pouca memória pode ser utilizado em dispositivos como *smartcards,* celulares e outros (TECHNOLOGY, 2001). Quanto maior o tamanho da chave de criptografia, maior irá ser a segurança fornecida pelo algoritmo, pois irão ser feitos mais turnos do AES (10, 12 e 14 respectivamente). Tendo em vista essa característica do algoritmo, para esse trabalho foi adotada uma chave de 256 *bits*. Para criptografar, cada turno do AES, com exceção do último, opera sobre um arranjo bidimensional de bytes com quatro colunas e quatro linhas (matriz de estados).

Os estágios do AES são subdivididos em **AddRoundKey**, onde cada *byte* da

matriz de estados é combinado com a subchave própria do turno (**RoundKey**) e cada subchave é derivada da chave principal usando o algoritmo de agendamento de chaves; **SubBytes**, que é constituído por substituição não linear onde cada byte é substituído por outro de acordo com uma tabela de referência (representada na Figura 2), na qual os primeiros bits do byte a ser substituído indexam a linha da tabela de substituição e os quatro últimos bits indexam a coluna ; **ShiftRows**, que consiste em uma transposição onde cada fileira do estado é deslocada, à esquerda, de um determinado número de posições de acordo com seu índice de linha da matriz de estado, ou seja, a linha zero irá ser deslocada em zero posições, a linha um e uma posição, e assim sucessivamente conforme pode ser notado na Figura 3; **MixCollumns**, que consiste de uma operação de mescla que opera nas colunas do estado e combina os quatro bytes de cada coluna usando uma transformação linear (Figura 4). O último turno substitui o último estágio de **MixCollumns** por um novo estágio de **AddRoundKey**. O funcionamento do AES está ilustrado na Figura 1.

Figura 1 - Caminho percorrido pela matriz de estados



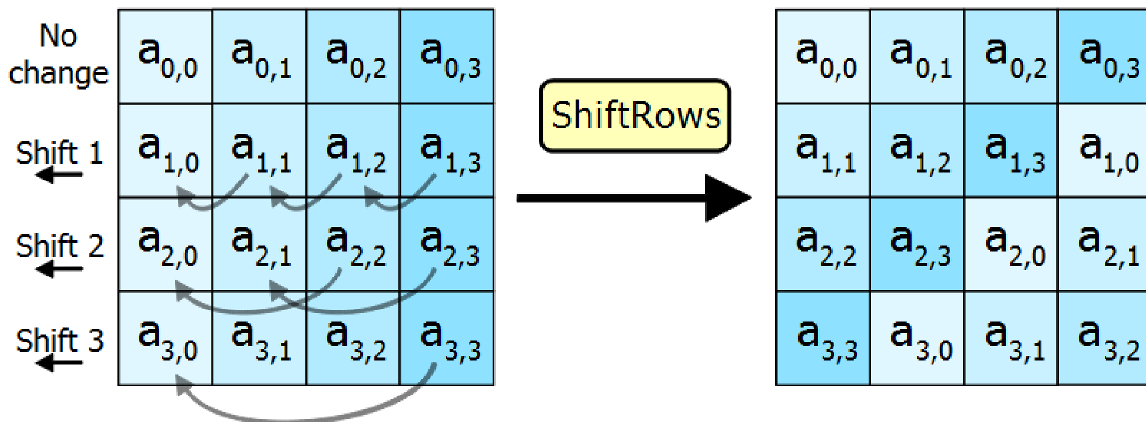
Fonte: Martins (2013).

Figura 2 - Tabela de referência para substituição

hex		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

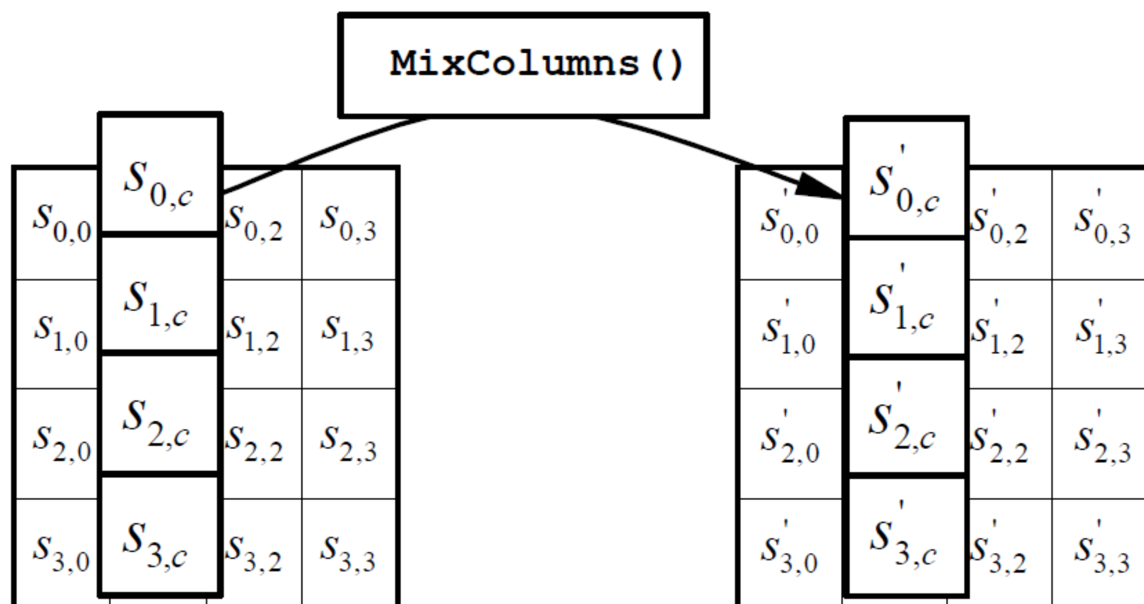
Fonte: Martins (2013).

Figura 3 - Funcionamento do *ShiftRows*



Fonte: Martins (2013).

Figura 4 - Funcionamento do *MixColumns*



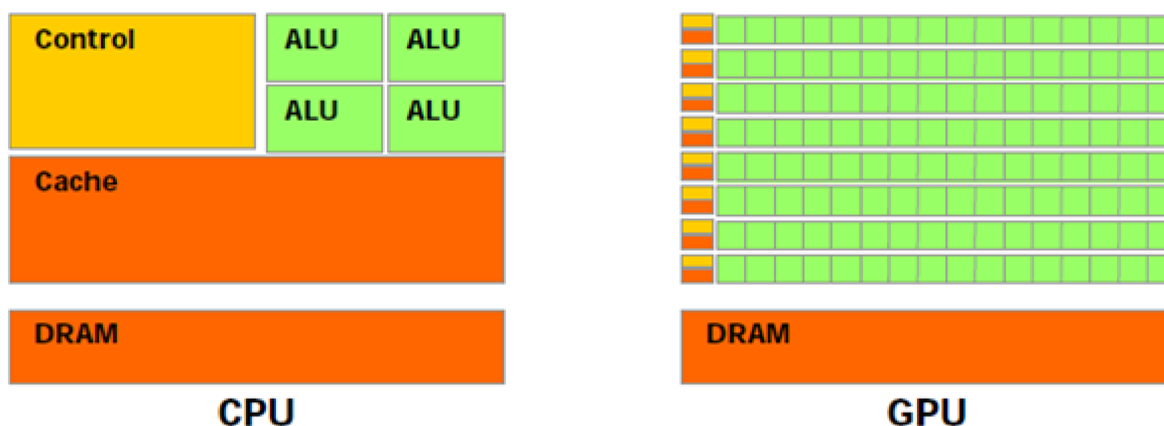
Fonte: Martins (2013).

2.2 GPU

As *Graphics Proccessing Units* (presentes nas placas de vídeo), tradicionalmente, são utilizadas para fornecer grande vazão de processamento gráfico requisitado pela crescente demanda dos usuários por alta qualidade de imagem em jogos, vídeos e demais aplicações multimídia. As GPUs têm sido objeto de estudos aprofundados durante os últimos anos e vem sendo utilizadas em aplicações de propósito gerais, fora da área de processamento gráfico. Essa difusão do uso de GPUs se deve a popularização da programação de propósito geral GPGPU desses dispositivos em função das respectivas APIs estarem se consolidando, como é o caso do CUDA (2011). As próximas subseções apresentam o funcionamento geral das GPUs e as arquiteturas das duas GPUs utilizadas nos experimentos.

As GPUs são especializadas em grandes quantidades de cálculos com precisão de ponto flutuante e em computação paralela - do que exatamente a renderização de gráficos se trata – então projetadas de tal modo que mais transistores são aplicados a processamento de dados ao invés de cacheamento de dados e controle de fluxo como é feito nas CPUs. Uma abstração sobre essa diferença arquitetural pode ser vista na Figura 5.

Figura 5 – Diferença arquitetural entre CPU *multicore* e GPU

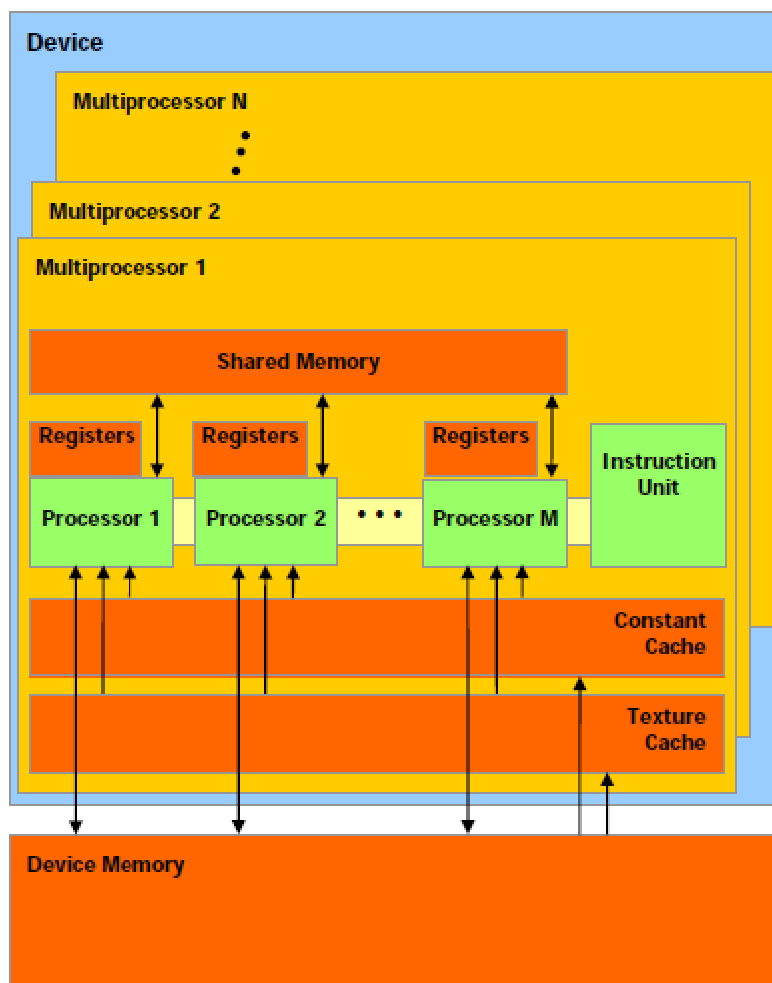


Fonte: NVIDIA (2012).

As GPUs fornecem melhor desempenho quando os problemas podem ser mapeados em aplicações onde a computação é feita com paralelismo de dados – como o modelo SIMD (*Single Instruction Multiple Data*) onde a mesma instrução é executada em muitos elementos de dados em paralelo – e, principalmente, onde a aplicação é predominantemente aritmética (NVIDIA, 2012). Uma vez que a mesma instrução é executada em cada elemento do dado, há uma pequena necessidade por controle de fluxo sofisticado e a latência de acesso a memória global do dispositivo pode ser camuflada com processamento em massa ao invés de grandes *caches* de dados.

A Figura 6 traz uma abstração das subdivisões e unidades do *hardware* de uma GPU, onde há os multiprocessadores e os níveis de memória. Dentro de cada multiprocessador há diversos elementos de processamento, registradores e memórias compartilhadas entre esses PEs, onde a quantidade deles depende da arquitetura do dispositivo.

Figura 6 – Modelo de *hardware* de uma GPU



Fonte: NVIDIA (2012).

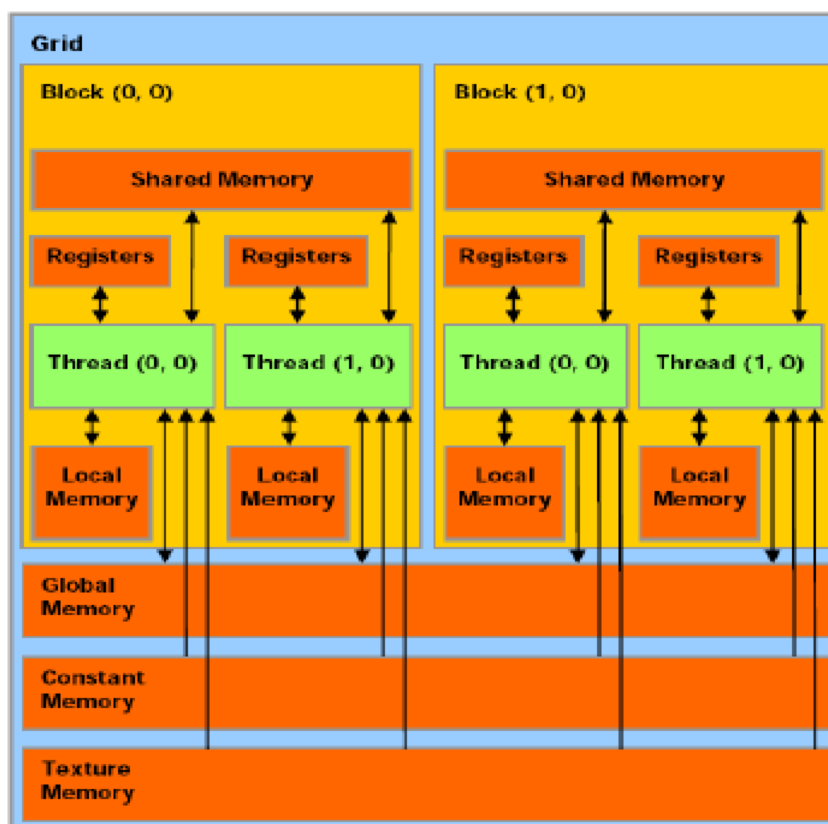
As GPUs possuem seis tipos de memória (NVIDIA, 2011), descritos a seguir e ilustrados na Figura 7.

- ***Registers***: são os registradores dos elementos de processamento que podem ser lidos e escritos pelas *threads*.
- ***Shared Memory***: pode ser lida e escrita pelas *threads* de um mesmo *block* (terminologia para divisão de grupos de *threads* na GPU). Esta memoria é dividida em *banks*. Se uma *thread* fizer acessos de leitura/escrita em *x* endereços de memória, caso todos estiverem localizados em diferentes *banks*, todas os acessos serão servidos simultaneamente, fornecendo uma grande vazão de dados. Contudo, se houver mais de um endereço contido em um mesmo *bank*, gerando um conflito, o acesso a estes endereços passa a ser serializado, reduzindo a largura de banda. Caso não haja conflitos, a *shared memory* é tão rápida quanto os registradores.
- ***Constant Memory***: quantidade de memória do dispositivo que só permite

leitura, sendo otimizada para esta operação. Pode ser lida por todas as *threads*.

- ***Global Memory***: é a memória principal da GPU, que pode ser lida ou escrita por todas as *threads* de um *grid*, além de ser usada como meio de comunicão com a CPU (host). Embora processador da GPU possa acessar diretamente qualquer dado na memória global, os dados irão ser transportados, em muitos casos, para a *shared memory* e os registradores no inicio do processamento porque a latência da *global memory* é muito alta.
- ***Local Memory***: memória local de uma *thread*.
- ***Texture Memory***: memória destinada ao armazenamento de texturas. Assim como a *constant memory,* somente operações de leituras são permitidas.

Figura 7 – Modelo de memória de uma GPU



Fonte: NVIDIA (2012).

A seguir serão apresentadas as arquiteturas das famílias de GPUs da segunda (GT200) e terceira geração (Fermi) que descenderam da G80 (NVIDIA, 2011a), pois elas foram utilizadas nos experimentos a GTS 250 (da família GT200) e a C2075 (da família Fermi).

Em junho de 2008, a NVIDIA introduziu uma revisão maior na arquitetura G80, produzindo a segunda geração da arquitetura unificada, a GT200. Em relação a primeira geração, a GT200 aumentou o número de *streaming processor cores* (posteriormente chamados de CUDA *cores*) de 128 para 240. Cada registrador teve seu tamanho dobrado, permitindo um maior número de *threads* sendo processadas simultaneamente. Para melhorar a eficiência no acesso à memória, foi adicionado o *hardware memory access coalescing*, responsável por camuflar os acessos devido à grande quantidade de *threads* e sistema de escalonamento melhorado, mantendo os CUDA *cores* ocupados com outros cálculos enquanto um dado é acessado na memória global do dispositivo. Também foi adicionado suporte à precisão dupla de ponto flutuante, a fim de dar suporte a aplicações científicas e aplicações de alta performance que necessitassem maior precisão.

A família G80 foi a visão inicial do que uma GPU com suporte facilitado a propósito geral deveria ter. A família GT200, por sua vez, ampliou a performance e as funcionalidades da G80. Já arquitetura Fermi representa um dos avanços mais significativos na arquitetura das GPUs desde a original G80. A Fermi foi projetada a partir de tudo que a NVIDIA havia aprendido de suas duas famílias de GPUs e de todas as aplicações que foram escritas para ela, empregando uma a abordagem diferente para criar a primeira GPU computacional do mundo (NVIDIA, 2011b).

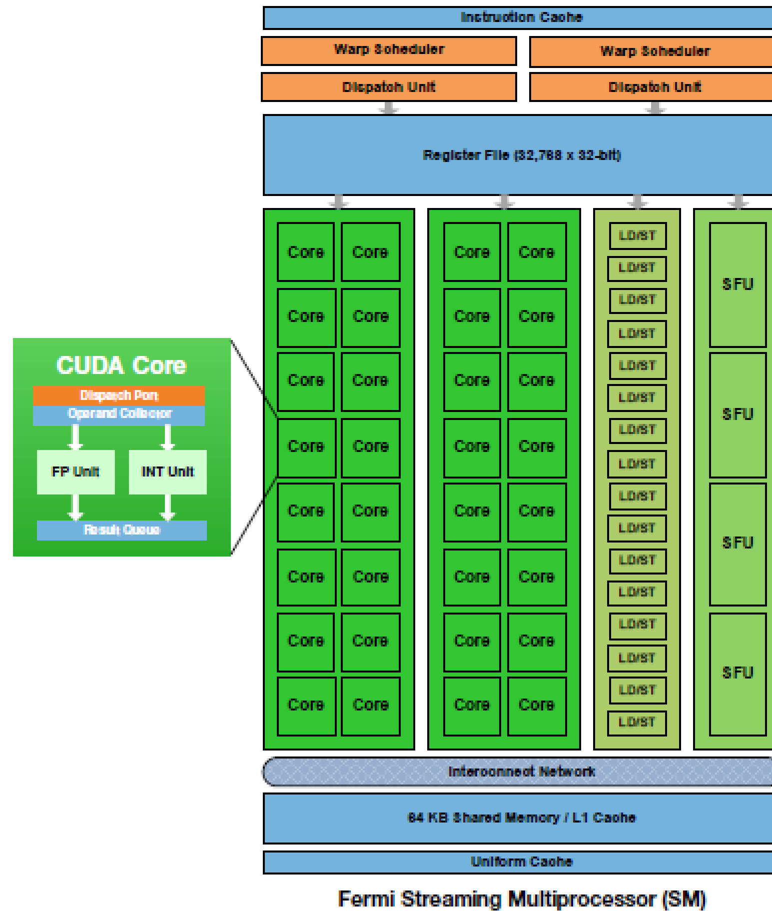
A Fermi apresenta as seguintes melhorias:

- Precisão dupla de melhor performance
- Suporte à código de correção de erro (*error-correcting code - ECC)*
- Hierarquia *true cache*
- Mais memória compartilhada *(Shared Memory)*
- Troca *de* contexto mais rápida
- Operações atômicas mais rápidas
- Escalonador *Dual Warp*

A primeira GPU da família Fermi foi implementada com três bilhões de transistores e até 512 CUDA *cores*. Um CUDA *core* executa uma instrução de ponto flutuante ou de número inteiro por *clock* para uma *thread*. Os 512 CUDA *cores* estão organizados em 16 *Simultaneous Multiprocessors (*SMs*)* de 32 núcleos cada. A GPU tem seis partições de memória de 64 *bits*, para uma interface de memória de 384-*bit*, suportando um total de até 6 GB de memória DRAM GDDR5. A interface com o *host*

é via *PCI-Express*. O *Gigathread Scheduler* distribui os blocos de *thread* para os escalonadores locais nos SMs. A Figura 8 ilustra a arquitetura de uma GPU da família Fermi.

Figura 8 – Arquitetura da família Fermi



Fonte: NVIDIA (2011b).

Tabela 1 – Principais diferenças entre as famílias de GPUs

GPU	G80	GT200	Fermi
Transistors	681 *million*	1.4 *billion*	3.0 *billion*
CUDA Cores	128	240	512
Double Precision Floating Point Capability	*None*	30 FMA *ops/clock*	256 FMA *ops/clock*
Single Precision Floating Point Capability	128 MAD *ops/clock*	240 MAD *ops/clock*	512 FMA *ops/clock*
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	*Configurable* 48 KB or 16 KB
L1 Cache (per SM)	*None*	*None*	*Configurable* 16 KB or 48 KB
L2 Cache	*None*	*None*	768 KB
ECC Memory Support	No	No	Yes

Fonte: NVIDIA (2011b).

2.3 APIs de programação paralela

Em 1965 Gordon Moore, fez uma estimativa de que o número de transistores que podem ser integrados em um único *die* dobra de 18 a 24 meses. Ele fez uma previsão de que a tecnologia de semicondutores irá dobrar sua eficiência a cada 18 meses. A frequência dos processadores também dobrou a cada ano, até chegar em um ponto em que a dissipação de calor seria tão alta que não seria energética e economicamente viável. Ao invés disso atualmente os fabricantes de microprocessadores têm aumentado o número de elementos de processamento (*PEs*) e diminuindo o *clock*, onde o ganho no desempenho se dá através do

paralelismo. A programação paralela cumpre um papel importante para o aumento do desempenho nas atuais arquiteturas que possuem dois ou mais *cores* (*multicores* ou GPUs), disponibilizando fluxos de computação individuais (*threads*), os quais serão processados pelos PEs.

Na programação paralela, são geradas *threads* criando fluxos paralelos de execução. Dependendo das características de um programa paralelo, as *threads* precisam ter um mecanismo que as gerencie, seja por troca de mensagens entre as *threads* ou por memória compartilhada, de tal modo que os acessos de escrita em memória por uma *thread* não invalidem os resultados gerados por outra *thread*. Para resolver esse tipo de problema, técnicas de sincronização de *threads* são adotadas ao se projetar um algoritmo paralelo, como no modelo produtor/consumidor que foi utilizado por este trabalho (GOMES, 2012c).

Os modelos de algoritmos paralelos são maneiras de estruturar estes algoritmos através de técnicas como decomposição, estratégias de minimização de interações entre os processos e seus respectivos mapeamentos. Para o modelo de produtor/consumidor é necessário um meio de comunicação, geralmente um *buffer*, onde o produtor gera dados para o consumidor processá-los. Para garantir a validade dos dados presentes no *buffer,* pode ser utilizado um *mutex* (TANENBAUM, 2010), que atua como um mecanismo de trava, garantido que somente um processo acesse os dados protegidos. Boas práticas de programação paralela requerem que o *mutex* seja sempre destravado após o acesso de um processo, a fim de evitar *deadlocks* (no qual os processos nunca terminam sua execução e os recursos do sistema ficam retidos, impedindo que outras tarefas sejam iniciadas).

Com a expansão da programação paralela em função das arquiteturas multiprocessadas, desenvolver aplicações paralelas e escaláveis se torna cada vez mais fundamental para explorar os recursos dessas arquiteturas. A criação de APIs que facilitem a codificação de tais aplicações se fez necessária para garantir que elas cheguem ao mercado em tempo hábil. Para este trabalho foram exploradas duas APIs voltadas a CPUs *multicores* (OpenMP e Pthreads) e uma API voltada a GPGPU (CUDA) que serão apresentadas nas subseções a seguir.

2.3.1 APIs para CPUs multicores

O OpenMP (*OpenMP-Multiprocessing*) é um modelo de programação para CPUs *multi-threaded* com memória distribuída, tornando explícito o paralelismo existente em programas, foi definido por um consórcio de fabricantes de *hardware* e *software*, juntamente com universidades e membros do governo norte-americano (OPENMP, 2013). Foi projetado para ser uma API escalável e portável, adicionando uma camada de abstração acima do nível das *threads*, deixando o programador livre da tarefa de criar, gerenciar e destruir *threads*.

O programador pode explicitar os locais do programa que serão paralelizados utilizando a diretiva #pragma disponível na linguagem. A etapa de pré-compilação do programa converte as diretivas em chamadas para *threads* comuns, como faz a API Pthreads por exemplo. O programador também não precisa usar as APIs de *threads* diretamente, o OpenMP fica encarregado de fazer isso automaticamente, trazendo portabilidade aos sistemas desenvolvidos, pois mesmo as APIs de *threads* mais abrangentes costumam apresentar diferenças em arquiteturas diferentes. Também é possível determinar junto ao sistema o número de núcleos ou processadores disponíveis. Assim, um mesmo código em OpenMP pode ser aproveitado em um sistema com um, dois, quatro ou mais *cores*, sem precisar ser adaptado, cabendo ao compilador adaptar o código existente para criar o maior número de *threads* possível.

Já a Pthreads é uma das APIs mais tradicionais na área de programação paralela (BUTENHOF, 1997). Ela fornece recursos de baixo nível de abstração para gerenciamento de *threads* através de rotinas que trabalham diretamente com as threads, criando, individualizando fluxos de computação, reduzindo (quando as *threads* terminam sua computação), além de incluir funções para designar valores para os atributos das *threads* (*joinable*, *scheduling,* etc) e funções de manipulação de *mutexes,* permitindo cria-los, destruí-los, trava-los e destrava-los. A Pthreads requer um cuidado adicional dos programadores, pois eles são responsáveis pela maior parte do gerenciamento dos recursos sistema quando estão utilizando *threads* criadas diretamente pela Pthreads.

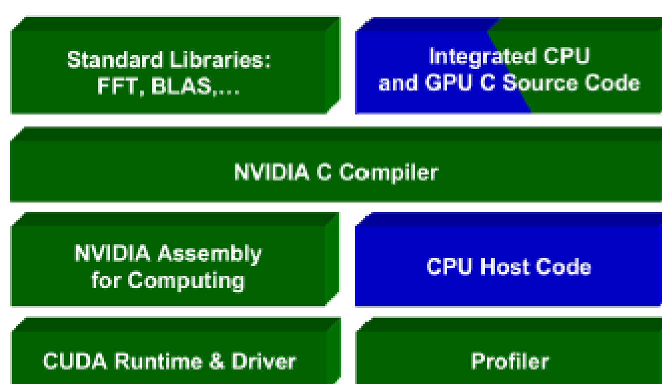
2.3.2 API GPGPU (CUDA)

As APIs GPGPU tem se consolidado nos últimos anos com a aceitação e popularização do uso de GPUs em aplicações de propósito geral como um *hardware* acelerador de desempenho. O CUDA é o exemplo mais consolidado desse tipo de API, em termos de recursos e documentação, o que levou a escolha dessa API para implementação do módulo de recriptografia em GPU.

O CUDA (2011) é uma plataforma de computação paralela e modelo de programação que fornece melhoria no desempenho por explorar a capacidade de processamento das GPUs. Desde sua apresentação em 2006, o CUDA tem sido amplamente desenvolvido através de milhares de aplicações em áreas como astronomia, biologia, química, física, mineração de dados; e artigos de pesquisa publicados, em mais de 300 milhões de GPUs com suporte a CUDA em *notebooks*, *workstations*, *clusters* e supercomputadores.

Essa API fornece um mecanismo de abstração de *hardware* que esconde os detalhes da GPU dos programadores, fornecendo portabilidade e escalabilidade às aplicações. Basicamente, o CUDA oferece um compilador C (*NVIDIA C Compiler*– NVCC) que gera código alvo tanto para o processador principal (*host*) como para a GPU (*device*) (Figura 9).

Figura 9 – Esquemático do NVCC

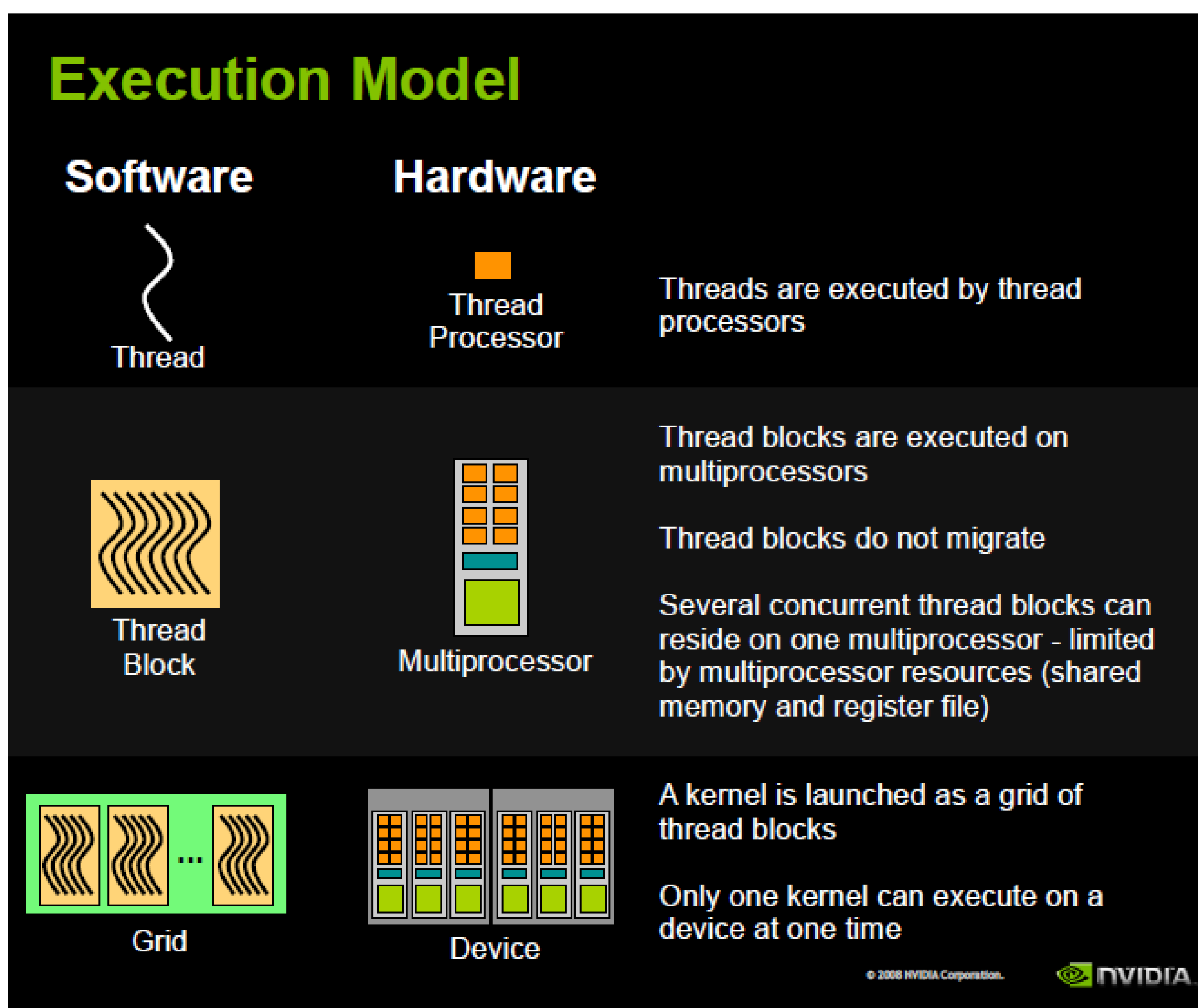


Fonte: NVIDIA (2012).

O processamento feito pela GPU é executado por conjuntos de *threads*, geralmente compostos por grupos de 32 *threads*, denominados *warps*. Cada *warp* executa em um dos SMs (*multiprocessors*), que compartilham entre si uma pequena memória (*shared memory*). Dessa forma, é possível atribuir a cada *thread* uma

pequena quantidade de dados e uma função principal (*kernel*) para operar sobre estes dados. Ao ser lançado, um *kernel* cria um *grid* de blocos de *threads que irão ser processados pelos CUDA cores* situados dentro dos SMs, como pode ser visto na Figura 10. Geralmente, quanto maior o número de *threads*, respeitando o limite imposto pela arquitetura da GPU alvo em contraste com o modelo de programação paralela utilizado pela aplicação, maior será o desempenho fornecido pela GPU (*device*). Tendo em vista a quantidade massiva de *threads*, o gerenciamento das mesmas é implementado em *hardware* (*Thread Execution Manager*) (NVIDIA, 2012).

Figura 10 - Abstração do modelo CUDA na GPU



Fonte: NVIDIA (2012).

3 TRABALHOS RELACIONADOS

Dentro do escopo deste trabalho, duas áreas de pesquisa se salientam: distribuição escalável de vídeo e aplicações de GPGPU envolvendo criptografia.

Como exemplo, em GRANADO (2010) é apresentada uma avaliação empírica de um protótipo para *proxies* de distribuição de vídeo baseado na técnica Memória Cooperativa Colapsada (CVCC), onde o foco é na análise sobre o consumo de largura de banda no enlace entre servidor e *proxy,* bem como na escalabilidade do sistema, medida em função da capacidade de atendimento de clientes concorrentes que é limitada pelo esquema de gerenciamento da memória.

Em YEUNG (2005) são apresentadas estratégias de recriptografia de vídeos em *proxies*, onde nem todos os segmentos do vídeo são cifrados devido à restrição de poder computacional disponível na época, deste modo obtendo uma vazão adequada ao tipo de serviço. Este trabalho se difere em relação à quantidade de segmentos cifrados, uma vez que todos os segmentos do vídeo são cifrados, protegendo integralmente o conteúdo, e não parcialmente.

Em SESHADRINATHAN (2008), os autores implementaram uma versão do AES para GPU utilizando API HLSL (*high level shader language*), que, hoje em dia, não é a mais adequada para GPGPU, por se tratar de uma API voltada a processamento gráfico. Os resultados desse trabalho demonstram que o uso de GPUs aplicado à criptografia é viável.

Em MANAVSKI (2007), foi feita uma comparação de desempenho entre uma implementação do AES na API OpenGL (*open graphics library*) e outra na API CUDA. Os autores também compararam essas implementações com uma versão de referência sequencial do AES executada em uma CPU, no qual os resultados demonstram que a API CUDA é mais eficiente para criptografia.

Já no artigo de DI BIAGGIO (2009) foram apresentadas duas estratégias de paralelização do AES, uma que abrange uma granularidade mais fina, entre as rodadas do algoritmo, e outra em granularidade mais grossa, que foca em um paralelismo de mais alto nível exposto pelos modos de operação *Electronic code book* (ECB) e *Counter* (CTR), além de uma análise do custo/benefício do uso de GPUs para criptografia. Apesar de ter como base de referência esse artigo (DI BIAGGIO, 2009), este trabalho se difere dos mencionados no contexto de criptografia em relação aos níveis de granularidade e a quantidade de

implementações diferentes, constituindo um estudo complementar e ao mesmo tempo mais aprofundado deste nicho de projeto.

Já em relação aos trabalhos sobre *proxy* de vídeo, representa um complemento aos módulos de gerência de memória propostos, atacando o problema da proteção do conteúdo que em geral não é abordado em trabalhos que visam avaliação de desempenho de sistemas escaláveis.

Neste cenário, cabe também destacar que não foram encontrados trabalhos que explorassem GPGPU em sistemas de distribuição de vídeo baseados em *proxies*. Por outro lado, conforme mencionado na introdução, este trabalho pode ser considerado uma extensão dos trabalhos anteriores (GOMES, 2011;GOMES, 2012a; GOMES, 2012b; GOMES, 2012c) mas com dois novos e diferentes enfoques específicos:

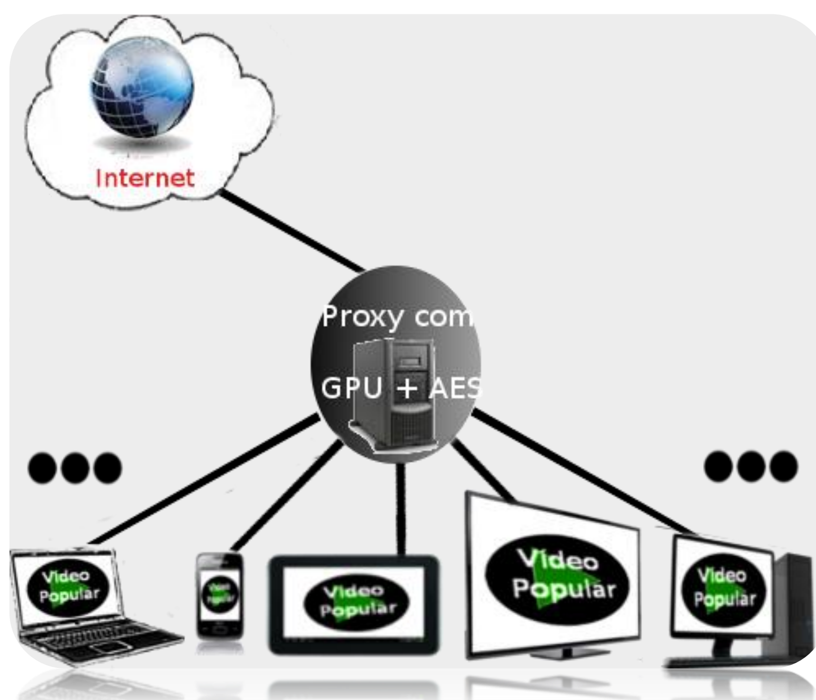
1. A criação de um modelo matemático (analítico) para estimativa do desempenho fornecido por uma GPU aplicada no contexto do módulo de recriptografia de vídeo sob demanda proposto pelos trabalhos anteriores.
2. Mensuração do consumo energético das diversas implementações do módulo de recriptografia, bem como a avaliação da solução de melhor custo-benefício para o *proxy* com o módulo de recriptografia.

4 RECRIPTOGRAFIA DE VÍDEO ASSISTIDA POR GPU

A partir dos conceitos básicos (Capítulo 2) e dos trabalhos relacionados (Capítulo 3), o presente capítulo detalha a aplicação objeto do estudo: a recriptografia assistida por GPU aplicada no contexto de distribuição escalável de vídeo. Além disso, são apresentados trabalhos anteriores que embasam a necessidade de um modelo analítico que permita generalizar a análise de desempenho de sistemas de distribuição baseados em recriptografia assistida por GPU, o qual é proposto e detalhado na sequência (Capítulo 5).

As soluções com *proxies* são as de melhor relação custo-benefício por complementarem a capacidade do servidor principal (que armazena de forma primária e não volátil os vídeos), por diminuírem a latência de início de exibição do vídeo e também por reduzirem a largura de banda necessária no enlace até o servidor (GRANADO, 2010). A Figura 11 ilustra um exemplo de aplicação de distribuição de vídeo com servidores *proxy*, apresentando um dentre vários proxies o qual estaria instalado na borda de rede para atender os usuários de um determinado sistema autônomo, sendo alimentado por um servidor primário localizado em algum ponto da Internet.

Figura 11 – Distribuição de vídeos usando proxies com recriptografia



Fonte: Gomes (2013).

Este módulo de recriptografia tem por finalidade proteger os direitos autorais correspondentes aos vídeos transmitidos pelo sistema, isto é, ele deve dificultar significativamente que os clientes não autorizados pelos detentores dos direitos autorais consigam ter acesso ao conteúdo protegido. Em períodos de alta demanda por acesso concorrente a vídeos populares, diversos sistemas escaláveis de distribuição realizam o agrupamento de clientes e o atendimento destes de forma sincronizada, ou seja, várias cópias do mesmo segmento de vídeo são criadas e enviadas aos vários clientes pertencentes ao grupo. Para aumentar a segurança do sistema DRM, é importante que os clientes não compartilhem uma mesma chave. Logo, durante a tarefa de criação das réplicas dos segmentos, é necessário recriptografar, para cada cliente, o conteúdo originalmente enviado de forma criptografada pelo servidor principal ao *proxy*, o qual fica armazenado desta forma por questões de segurança na *cache* local deste elemento intermediário do sistema de distribuição. Ou seja, este modelo envolve o uso de uma chave de criptografia diferente para cada um destes clientes, criando desta forma sequências de *bytes* individualizadas que serão enviadas em fluxos *unicast* para os clientes pela interface de rede do *proxy*.

Para criptografar os segmentos, foi escolhido como cifrador um dos algoritmos mais populares usados para criptografia de chave simétrica, o AES (DAEMEN, 2002), na sua variante conhecida como *Electronic Code Book* (ECB). Esta variante do algoritmo prevê um tamanho de bloco fixo de 128 *bits* (16 *bytes*, organizados em um arranjo bidimensional de *bytes* com quatro colunas e quatro linhas) e uma chave com tamanho de 256 *bits* (conforme explicado na Seção 2.4.2).

Em particular, o ECB possui um alto grau de paralelismo quando aplicado a criptografia de segmentos de vídeo, uma vez que não há dependência entre *slices* (resultantes da divisão dos segmentos de vídeo em fatias de 128 *bits*), possibilitando que cada fatia seja cifrada em concorrência sem a necessidade de sincronização intermediária.

As versões que fazem uso da GPU utilizam a API CUDA. Por outro lado, as versões que são executadas somente na CPU *multicore* também têm suas *threads* de criptografia criadas através da API Pthreads ou da OpenMP, competindo pelo reduzido número de núcleos de processamento quando comparado com as GPUs mais recentes.

A metodologia para o desenvolvimento do módulo de recriptografia seguiu os

passos descritos nas subseções a seguir.

Em Gomes (2009) foi proposto como ponto de início o isolamento do algoritmo de criptografia AES de sua biblioteca OpenSSL, evitando dependências desnecessárias para o módulo. Essa modularização proposta adotou os seguintes passos:

1. Verificar a dependência dos *headers* no núcleo do AES (arquivo aes_core.c);
2. Desfazer as dependências do item 1, com a cópia das funções e definições presentes nos arquivos *headers;*
3. Foi criado um programa de teste para checar se os itens anteriores foram atendidos.

Uma vez que o núcleo do AES foi isolado, foi feito o primeiro programa que faz uso da GPU GTS 250. Para isso o núcleo do AES foi codificado com um *kernel* CUDA que cifra um bloco de entrada com uma chave de 128 *bits.*

Após a validação do primeiro *kernel* desenvolvido, foi iniciado o processo da criação do módulo de recriptografia, a fim de criptografar um vídeo de entrada para *n* clientes. Para tal foram empregadas as técnicas de programação paralela modelo produtor/consumidor, sincronização de *threads* por meio de semáforos e *buffers* de entrada e de saída conforme será descrito a seguir.

Em primeiro lugar, foi esquematizado o funcionamento do modelo produtor/consumidor com adaptação para o módulo de recriptografia. Para tal foram implementadas, em linguagem C/C++ e em todas as versões do módulo, as *threads* de leitura e de controle através da API Pthreads. A *thread* de leitura é responsável por produzir segmentos do vídeo de entrada e coloca-los no *buffer* de entrada, já a *thread* de controle é responsável por consumir os segmentos do vídeo e gerenciar o núcleo de recriptografia (GPU ou *multicore*). Uma abstração do funcionamento geral do módulo de recriptografia pode ser visto na Figura 12, onde é demonstrado o mecanismo principal para a criptografia dos segmentos de vídeo nas versões que fazem uso da GPU. Para as versões *CPU-only* o funcionamento é análogo, diferindo no “prepara o contexto” que é substituído pela configuração das *threads* de criptografia, dependendo da versões OpenMP ou Pthreads, e também no “dispara o *kernel*” que é o momento em que são criadas essas *threads*.

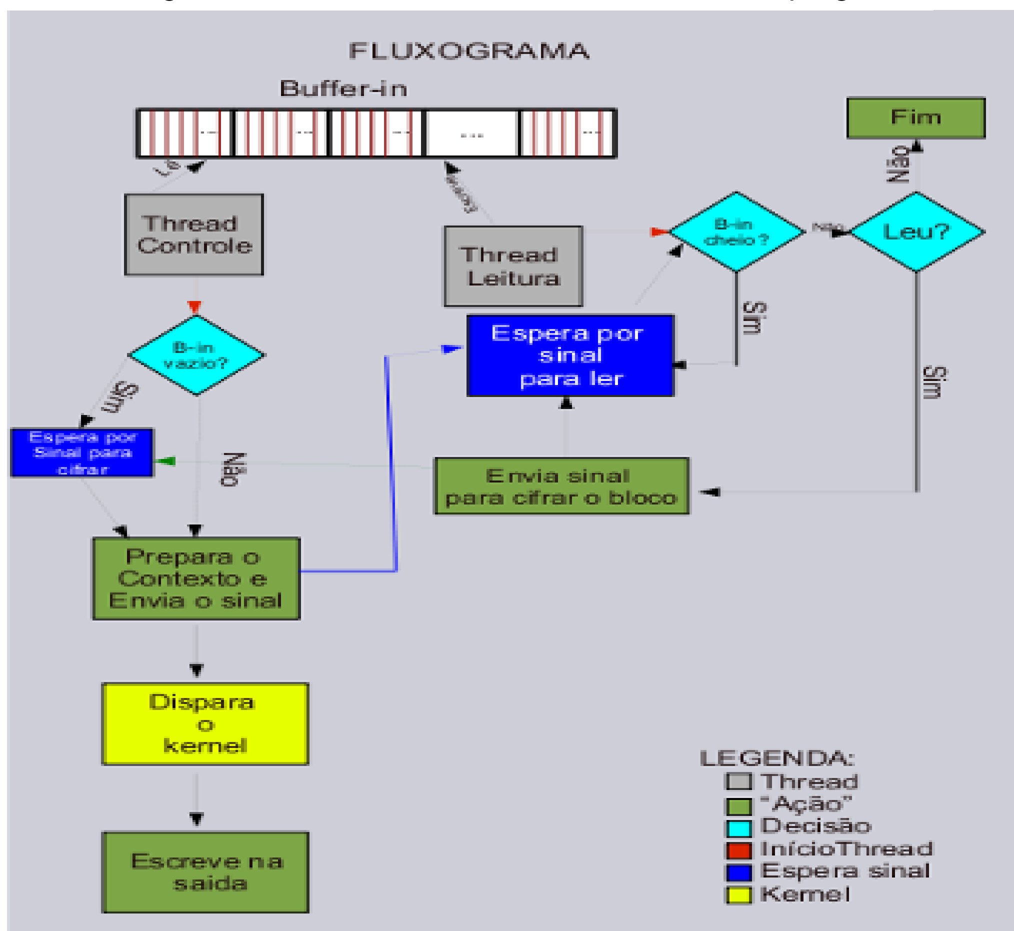
Uma vez que o módulo opera com um modelo produtor/consumidor, a *thread* de leitura está encarregada do papel de produtor, exclusivamente gerando os segmentos de vídeo, a partir do vídeo de entrada, e os depositando no *buffer* de

entrada. Para não haver inconsistência nos dados do *buffer* de entrada, foi utilizado um *mutex* para gerenciar os acessos de escrita e leitura. Quando a *thread* deposita um segmento no *buffer*, ela pede o *lock* ao *mutex* para realizar a escrita. Ao terminar a produção ela destrava o *buffer* e envia um sinal para a *thread* de controle despertar, informando que há pelo menos um segmento de vídeo para ser consumido no *buffer* de entrada.

Para completar o modelo produtor/consumidor, a *thread* de controle está encarregada de consumir os segmentos de vídeo depositados pela *thread* de leitura no *buffer* de entrada. A *thread* de controle também é responsável por carregar os parâmetros das *threads* de criptografia nas versões Pthreads e OpenMP. Além disso, na versão CUDA é ela que gerencia as cópias de memória (entre *host* e *device*) dos segmentos do vídeo de entrada, seta os parâmetros do *kernel* (número de *grids, blocks, threads* e quantidade de memória alocada na GPU).

Para que o modelo produtor/consumidor funcione eficientemente, foram criados dois *buffers*, um de entrada e outro de saída. Os *buffers* são utilizados para diminuir o acesso de leitura e escrita em disco, otimizando o desempenho do módulo de recriptografia. Eles foram implementados utilizando como bloco de construção uma estrutura de dados que contém um ponteiro para o segmento do vídeo e a identificação do cliente (*id)* do cliente, e para formação do *buffer* é utilizado um *array* dessa estrutura com o tamanho definido no número máximo de clientes (que pode ser configurado através do *header* das versões do módulo de recriptografia). Além desse *array*, se fez necessária as informações sobre o início e fim do *buffer* circular e se ele está ocupado, e elas estão condensadas em outra estrutura de dado.

Figura 12 – Fluxograma do funcionamento do módulo de recriptografia



Fonte: Gomes (2012c).

As informações sobre a quantidade de clientes e suas respectivas chaves de criptografia estão contidas em um arquivo texto que contem 32 caracteres por linha, onde cada linha corresponde a uma chave de um cliente. O vídeo de entrada utilizado na maioria dos experimentos tem tamanho de 4,8 MB. Também é possível configurar o número de clientes atendidos em paralelo (NCP), número de *threads* cooperativas (NCT), escrita direto em memória (ao invés de escrever no arquivo de saída) e o fator multiplicativo do tamanho do segmento do vídeo ao invocar o comando *make* na compilação das versões do módulo.

Uma vez que para os testes de simulação é necessário ter várias amostras de um mesmo cenário com mesma configuração, a automatização desse processo é crucial para a rapidez com que os resultados são gerados.

Os *scripts* criados para rodar em um ambiente Linux, usam os comandos de um terminal, tais como as funcionalidades de troca de diretório (cd), compilação através do Makefile (*make*) das versões do módulo de recriptografia, *loops* para

repetição (*for*) da configuração do cenário e a própria execução do simulador do módulo de criptografia e suas respectivas criação de arquivos de *log* para posterior análise.

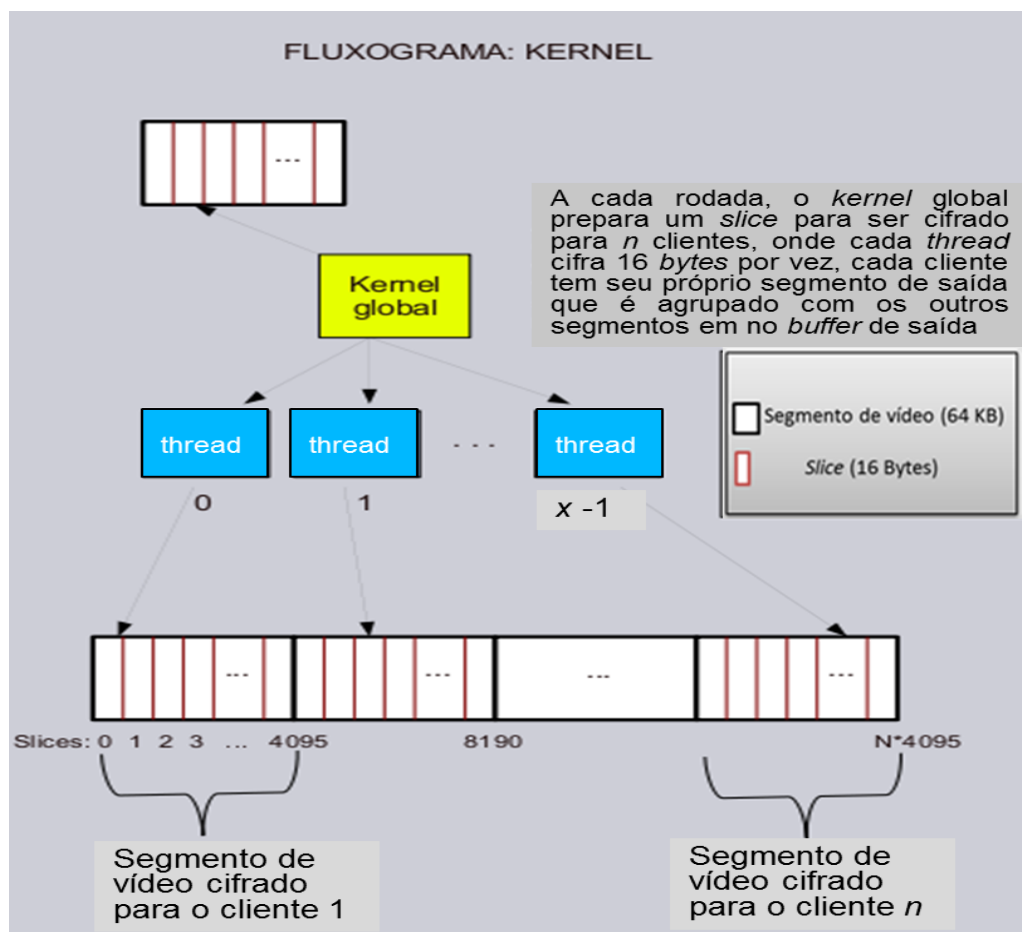
Para cada implementação do módulo, os cenários testados variaram as configurações de número de clientes, tamanho do vídeo e tamanho do segmento de vídeo.

A fim de avaliar comparativamente as variantes do AES, buscando a que melhor se integra com o sistema de distribuição proposto, foram produzidas três implementações principais, correspondentes a três níveis de paralelismo explorados, e outras três derivadas da última, sendo elas *coarse-grained*, *fine-grained,mix-grained* e *mix-grained adaptativa* e suas derivadas. Essas implementações serão detalhadas nas próximas subseções.

4.1 Coarse-grained

Com o propósito de atender (ou seja, criptografar um vídeo inteiro para) os *n* clientes, essa primeira implementação cria *n threads.* Cada uma delas é responsável por criptografar os segmentos do vídeo, o qual são produzidos pela *thread* de leitura, de um cliente. Esses segmentos por sua vez são subdivididos em fatias (*slices*) de 16 *bytes* para serem cifrados pelo núcleo do AES. Como cada cliente é atendido por uma *thread*, cada uma recebe uma chave de criptografia diferente. Na implementação dessa versão em CUDA, o núcleo do AES está dentro do *kernel*, que por sua vez, cria as *n threads* de criptografia. Para fins comparativos entre GPU e CPU *multicore* foi desenvolvida uma versão em Pthreads que é executada somente nos *núcleos* disponíveis no *multicore* do ambiente experimental. Os primeiros resultados com essa versão mostraram um desempenho inferior da GPU em relação a CPU, devido a carga de trabalho das *threads* relativamente grande (e em pequena quantidade) para os elementos de processamento da GPU (como explicado na seção 2.2). Em Gomes (2012a) foi criada uma implementação da *coarse-grained* com a API OpenMP. O funcionamento da versão *coarse-grained* está ilustrado na Figura 13.

Figura 13 – Fluxograma de funcionamento da versão *coarse-grained*

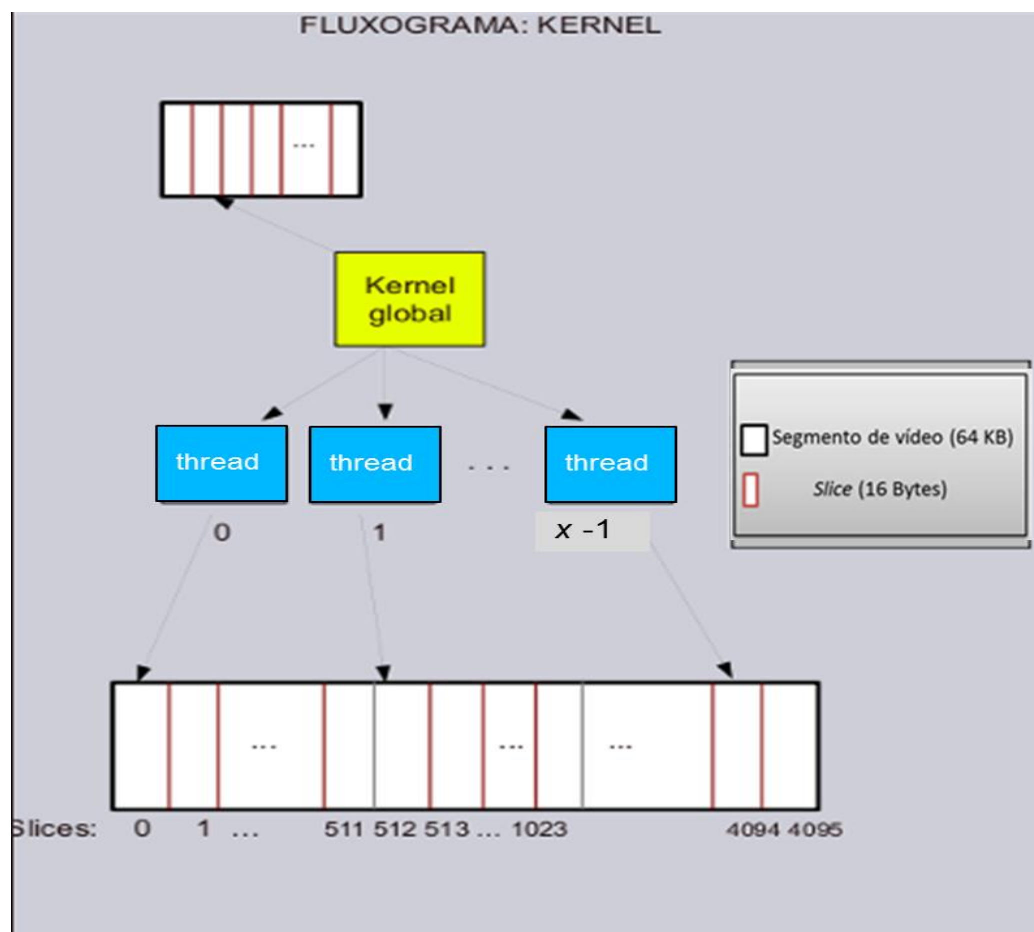


Fonte: Gomes (2012c).

4.2 Fine-grained

Uma vez que a versão *coarse-grained* implementada em CUDA não estava fornecendo desempenho satisfatório, foi feita uma nova abordagem do módulo de criptografia com uso de GPU. Tendo em vista que as GPUs fornecem mais desempenho em modelos de programação SIMD, na versão *fine-grained* são criadas *x threads* cooperativas para criptografar um segmento de vídeo inteiro para um cliente. Uma vez que há um grupo de *threads* cooperativas cifrando apenas um segmento do vídeo para um cliente, esse processo é serializado e repetido até que todos os *n* clientes tenham o vídeo criptografado. Mesmo com essa serialização no atendimento aos clientes, essa versão forneceu desempenho superior a versão *coarse-grained* em CUDA. O funcionamento da versão *fine-grained* está ilustrado na Figura 14.

Figura 14 – Fluxograma de funcionamento da versão *fine-grained*

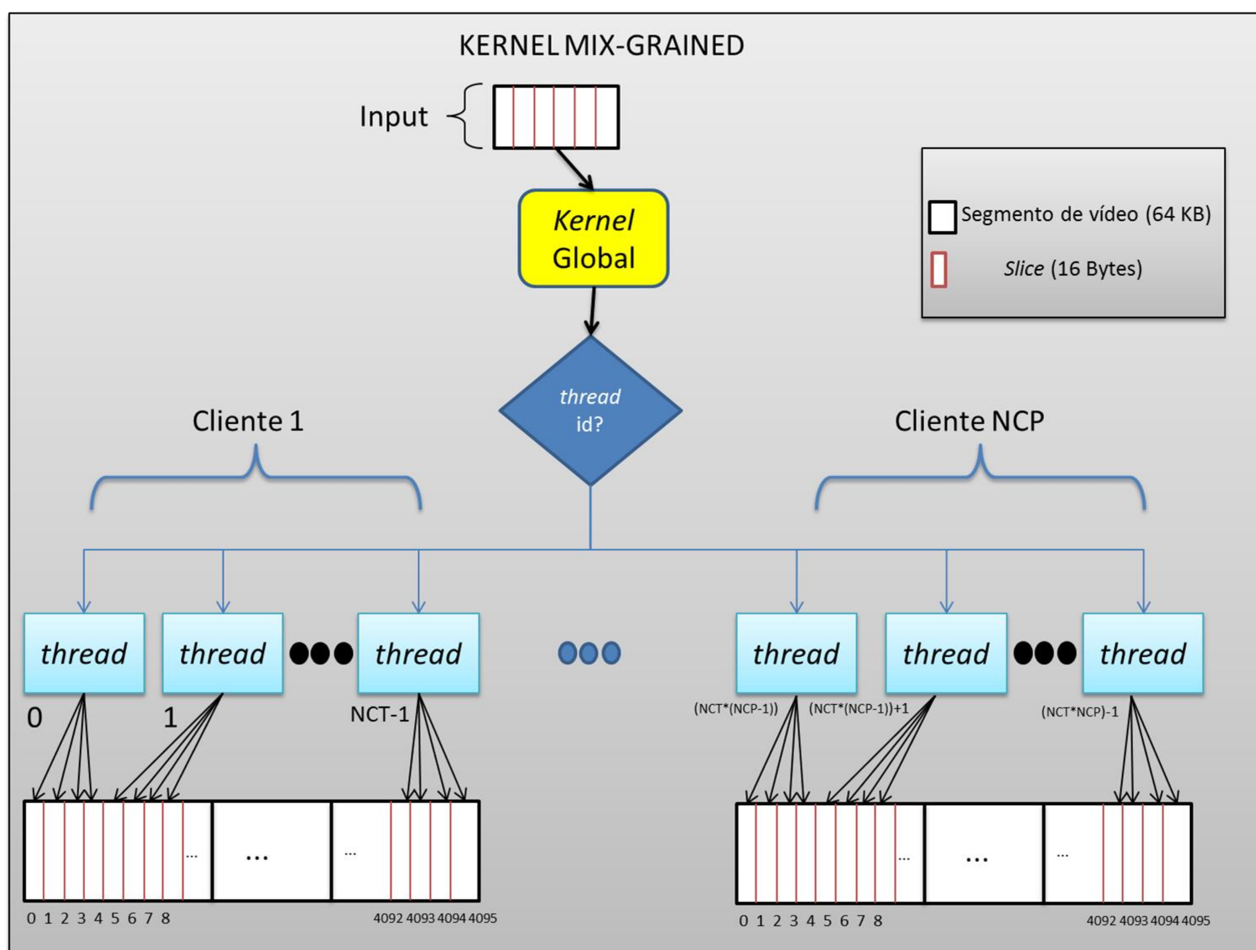


Fonte: Gomes (2012c).

4.3 Mix-grained

A *mix-grained* surgiu da ideia de por fim ao atendimento serial dos grupos de *threads* da *fine-grained* e assim melhor explorar a arquitetura da GPU. Essa versão é uma mescla da primeira com a segunda, onde um grupo de *threads* opera em cooperação para cifrar um segmento para um cliente, enquanto outros grupos de *threads* cooperam para atender outros clientes, de modo que cada cliente é representado por um grupo de *threads.* Conforme será demonstrado na seção dos resultados essa foi a versão divisora de águas do trabalho, pois através dela seria possível obter uma vazão de criptografia satisfatória no atendimento a uma grande demanda de clientes. O funcionamento da versão *mix-grained* está ilustrado na Figura 15.

Figura 15 – Fluxograma de funcionamento da versão *mix-grained*



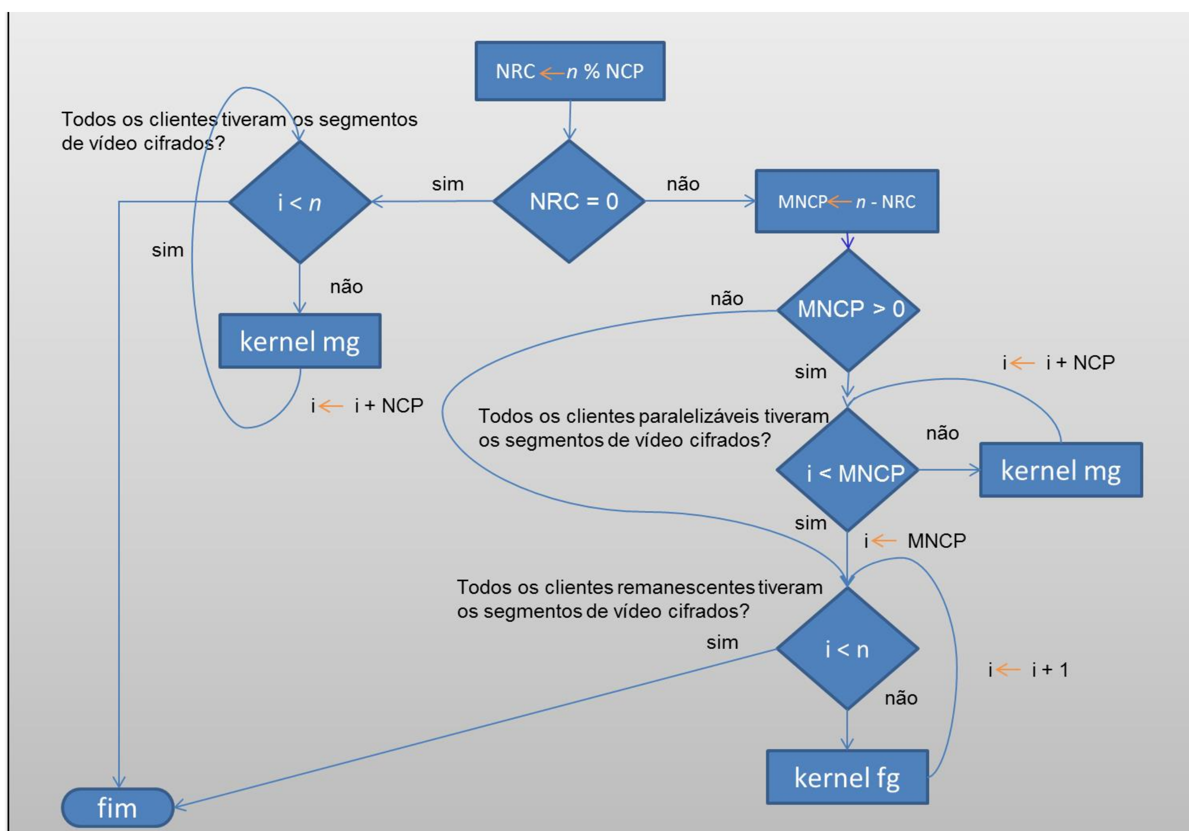
Fonte: Gomes (2012c).

4.4 Mix-grained adaptativas

A primeira versão *mix-grained* adaptativa foi desenvolvida para sanar uma restrição quanto ao número de clientes que requisitariam o mesmo vídeo, uma vez que só seria possível atender quantidades de clientes múltiplas de oito. Essa restrição foi em função do *best-fit* do NCP para a GPU GTS 250 que fez parte do primeiro ambiente experimental (o qual será detalhado na seção Ambientes experimentais). A *mix-grained* adaptativa foi implementada em três abordagens distintas (**A**, **B** e **C**). Conforme será demonstrado nas subseções adiante, no melhor caso (quando *n* é múltiplo de NCP) são criadas NCP*NCT (número de *threads* cooperativas, também é configurável) *threads* divididas entre NCP conjuntos de *threads*, cada um com NCT *threads* cooperativas. O *best-fit* do NCT é 1024 para as implementações que fazem uso de GPU, pois cada *thread* é responsável por cifrar quatro fatias do segmento de vídeo.

A abordagem **A** consegue atender uma quantidade flexível de clientes, comparando o NCP – configurado pelo *kernel*, sendo múltiplo de dois - com a carga de trabalho (*n*). Se *n* for múltiplo de NCP, os clientes são atendidos em concorrência de NCP a NCP até que todos sejam atendidos. Se *n* for maior e não múltiplo de NCP, é calculado a quantidade de clientes que podem ser servidos em concorrência e esses são atendidos no modelo *mix-grained*, a quantidade remanescente de clientes (NRC) é atendida, por simplicidade, no modelo *fine-grained*. Caso *n* seja inferior a NCP, todos os clientes são atendidos no modo *fine-grained*. O funcionamento da abordagem **A** *mix-grained* adaptativa está ilustrado na Figura 16.

Figura 16 – Fluxograma de funcionamento da abordagem A da *mix-grained* adaptativa

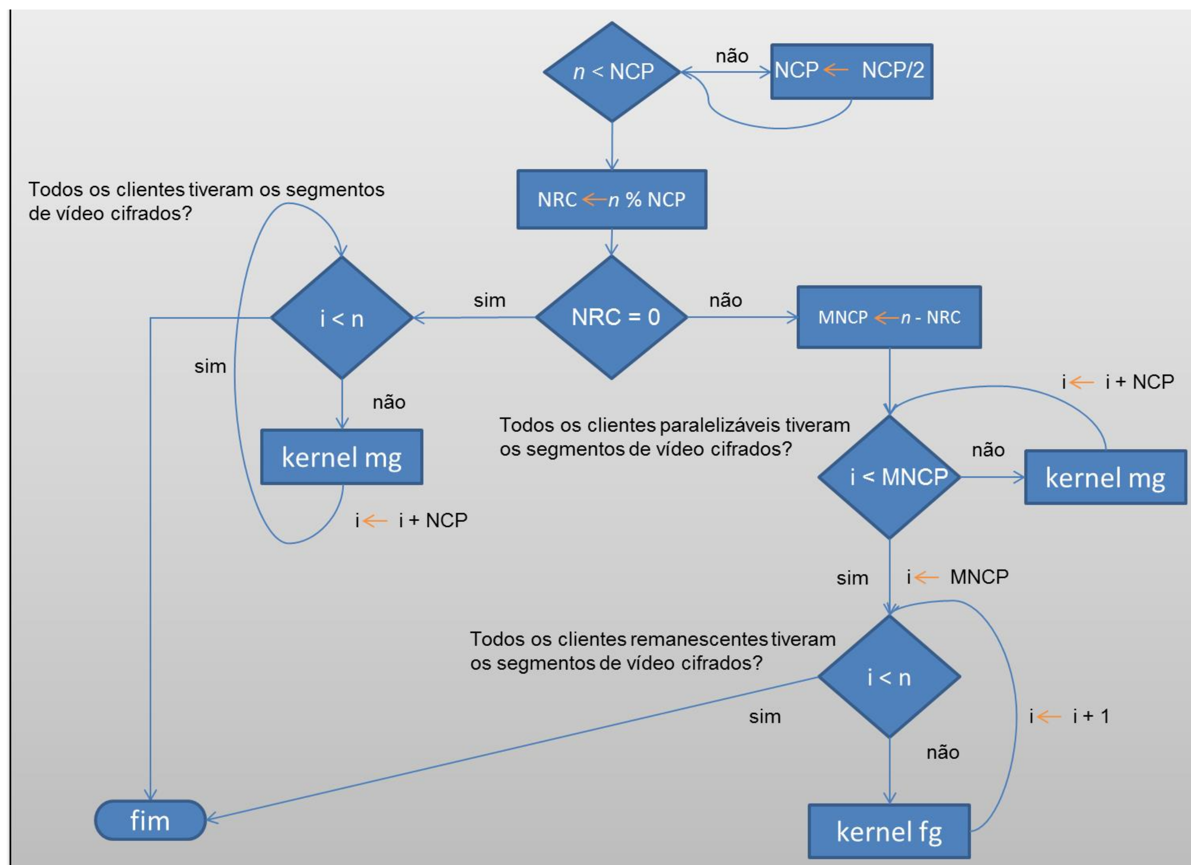


Fonte: Gomes (2012c).

Já a abordagem **B**, é semelhante a anterior, porém se *n* for menor que NCP, ele é divido por dois até que seja possível executar um *kernel* (múltiplo de dois) que atenda essa demanda, e assim sucessivamente até que reste um número *x* de clientes que não se enquadre nos *kernels mix-grained*, então os clientes remanescentes são atendidos serialmente no modelo *fine-grained*. O funcionamento

da abordagem **B** *mix-grained* adaptativa está ilustrado na Figura 17.

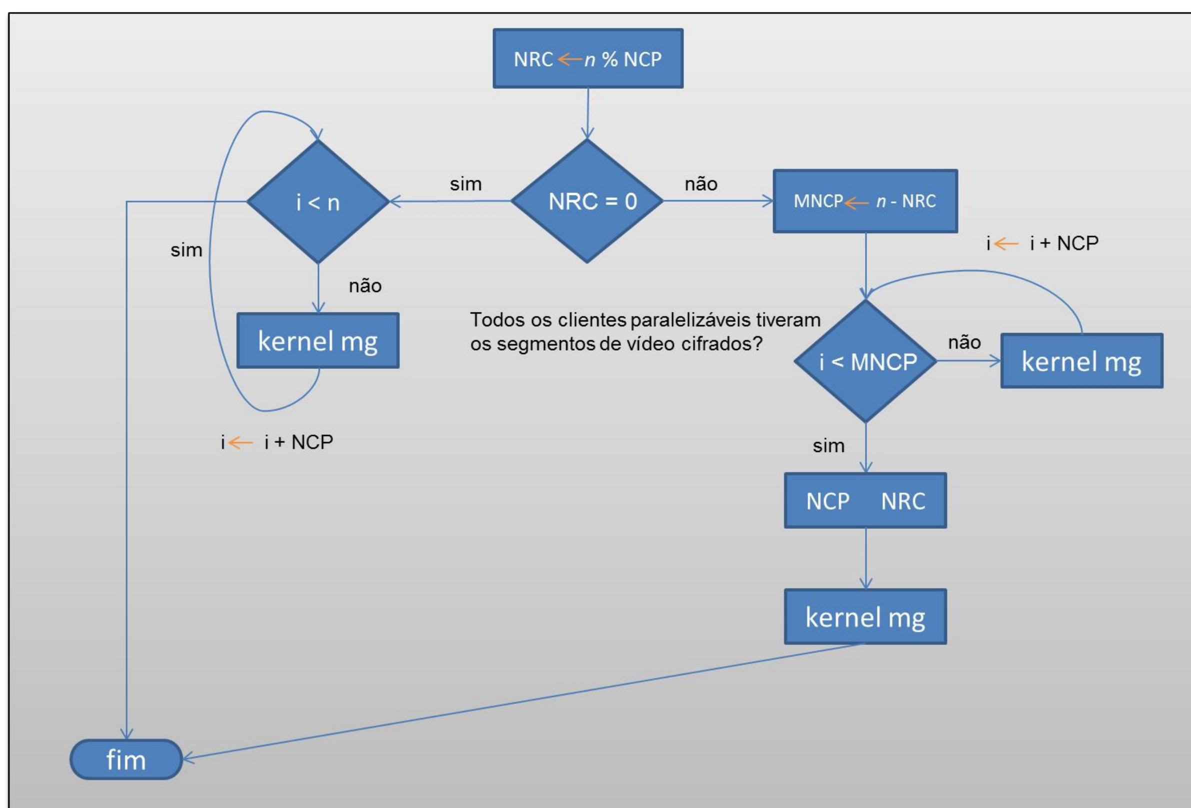
Figura 17 – Fluxograma de funcionamento da abordagem B da *mix-grained* adaptativa



Fonte: Gomes (2012c).

E por fim, na abordagem **C**, clientes remanescentes serão atendidos por outro *kernel* configurado dinamicamente a partir da quantidade clientes remanescentes, abolindo o uso da *fine-grained* por completo – conforme a Figura 18.

Figura 18 – Fluxograma de funcionamento da abordagem C da *mix-grained* adaptativa



Fonte: Gomes (2012c).

4.5 Métricas de avaliação

A seguir irão ser apresentadas as métricas, desempenho e consumo energético, para avaliação do custo-benefício das diversas implementações do módulo de recriptografia.

4.5.1 Análise de desempenho

Objetivo desta foi verificar a viabilidade e a eficiência do uso da GPU, em contraste ao emprego de um processador *multicore*, tomando por base uma métrica tradicional de avaliação do desempenho (tempo de execução, em milissegundos) e outra associada diretamente à aplicação (vazão de recriptografia, em Mbps).

Os experimentos foram realizados em dois ambientes de testes:

- Servidor HP Proliant ML350 G6: Processador Intel Xeon E5620 (4 *cores,* 2.4 GHz por *core, com Hyper-Threading*); **Memória RAM:** 8 GB DDR 3; **GPU:** nVidia Tesla C2075 6GB 384-*bits* (448 elementos de processamento); **Placa mãe:** HP 461081-001 395566-003**; Sistema Operacional:** Ubuntu 10.04 LTS (*kernel* 2.6.32-38-generic-pae);

- *Desktop*: Processador Intel Core i5 750 (4 *cores*, 2.67 GHz por *core*, sem *Hyper-Threading*); **Memória RAM**: 4 GB DDR3; **GPU**: nVidia GTS 250 512MB 256-*bits* (128 elementos de processamento); **Placa mãe**: Intel DP55WB; **Sistema Operacional**: Linux 32-*bits* Ubuntu 9.10 (*kernel* 2.6.31-14-generic).

Para construção de todos os cenários experimentais foram utilizados compilador GCC 4.4.3**,** CUDA SDK 4.2**,** CUDA toolkit 4.2**,** CUDA driver 295.41 e CUDA GDB 4.0**.**

Foram feitos experimentos para avaliar a carga de trabalho em função da demanda por desempenho em diversas configurações a fim de determinar a configuração de hardware e *software* que melhor se adequa a aplicação, seja com o uso de uma GPU ou apenas o uso de um *multicore*. Para tanto, foi realizada uma análise de sensibilidade com números crescentes de clientes simultâneos (*n*) com o intuito de simular picos de requisições a um vídeo popular (para cada configuração foram realizadas várias execuções onde a análise estatística dos tempos de execução – intervalo de tempo entre o início da criptografia do primeiro segmento de vídeo para o primeiro cliente e o final da criptografia do último segmento para o enésimo cliente – mostrou que não houve variação significativa dos resultados).
A partir do tempo de execução da simulação é possível extrair a métrica vazão que é dada por:

$$Vazão = \frac{Entrada\ (MB)*8*n}{Tempo\ de\ execução\ (s)}\ (Mbps) \quad (1)$$

Os experimentos consistiram em lançar cargas de trabalho (criptografia de segmentos de um vídeo para *n* clientes com chaves de criptografia diferentes) para diversas configurações do *proxy*. Para as versões que fazem uso da GPU, os *kernels* foram configurados com a versão adaptativa do *mix-grained*.

Cada teste foi executado dez vezes e foi feita uma média do tempo de execução de cada configuração, embora a variação entre execuções tenha sido insignificante. A carga de trabalho dos experimentos variou de 1 a 1024 clientes, sendo que de 1 a 512 para os testes mais específicos - contrastar a diferença de desempenho entre as diversas configurações, e de 1 a 1024 para as versões que demonstraram o melhor desempenho dentre as demais. Foi decidido não realizar os experimentos na GTS 250 com NCP maior que oito porque, quando isso acontece, a vazão vai decaindo conforme o NCP aumenta, ou seja, o limite para ganho no desempenho é atingido quando se tem NCP igual a oito nessa arquitetura. Além dos

experimentos práticos da aplicação foram feitas análises para testar a ocupação dos núcleos CUDA pelas *threads* e o impacto sobre a vazão fornecida para cifrar um vídeo inteiro para um cliente.

Nas avaliações foram usados três modos de execução:

- **GPU**: onde os *kernels* disparados são executados nos elementos de processamento da GPU**;**
- **Pthreads** e **OpenMP**: versões que criam *threads* concorrentes, até um limite máximo especificado por parâmetro, as quais são escalonadas pelo sistema operacional para execução concorrente sobre os núcleos da CPU.

Tabela 2 - Vazão demandada para streaming com diferentes padrões de compressão e perfis de vídeos

	MPEG-4 AVC vs. MPEG-2	
	MPEG-2	**MPEG-4 AVC (H.264)**
Talking Head (Steady content, low contrast)	2.2 - 4.0 Mbps	0.7 - 1.4 Mbps
Action Video (Dramatic changes, high contrast)	4.0 - 7.0 Mbps	1.6 - 3.0 Mbps
End-to-End Latency (ms)	90 ms (lowest hai500) 170 ms (average)	120-150 ms (hai1000)

Fonte: Haivision (2011).

Além dos resultados obtidos nos experimentos com as versões implementadas com CUDA, Pthreads e OpenMP, são incluídas linhas tracejadas que expõem uma estimativa do desempenho necessário para atender os clientes que estariam requisitando um vídeo a uma determinada taxa de exibição, como pode ser visto na Figura 19. Por exemplo, conforme a Tabela 2, para vídeos com baixa movimentação (*Talking Head*), quando usado o padrão de compressão MPEG-4 AVC (H.264), considerado o estado-da-arte atual, a vazão demandada por cada cliente é 1,4 Mbps no pior caso, enquanto que em vídeos de ação (*action video*), a demanda chega a 3 Mbps. Neste caso, a vazão demandada pelo *proxy* para 512 clientes simultâneos superaria a capacidade efetiva da interface de rede, assumindo que esta seria usualmente de 1 Gbps, embora esteja sendo cada vez mais difundido o uso de interfaces de 10 Gbps em aplicações de alta demanda por largura de banda.

Conforme pode ser observado na Figura 20, as curvas seguem dois

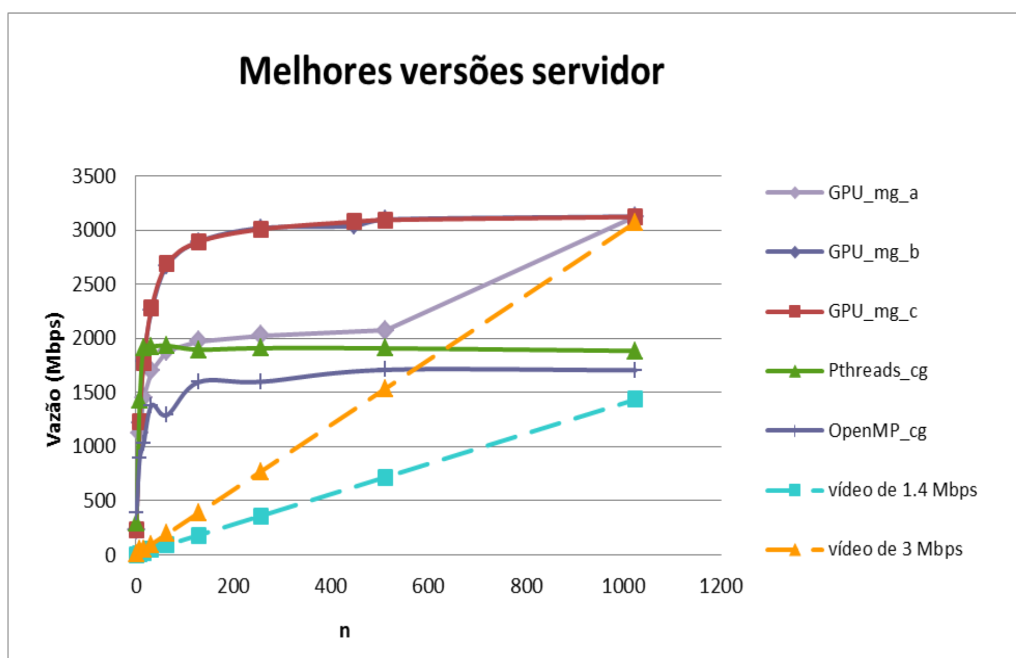
comportamentos distintos para a abordagem **A**, conforme a relação entre o NCP e o número de clientes. Em cenários onde o número de clientes (*n*) é inferior ao NCP configurado, as abordagens **B** e **C** fornecem uma vazão em média 23,8% melhor que abordagem **A** e para 512 clientes com um NCP igual a 1024, que é o pior caso, 33,2% melhor. Isso ocorre porque na abordagem **A** (nesse caso), o *kernel* executado é o *fine-grained*.

A Figura 20 resume os resultados das configurações de melhor desempenho das diferentes implementações mensurados no servidor. Com o aumento da carga para 1024 clientes, apenas as versões que exploram a GPU atendem à vazão demandada para vídeos MPEG-4 AVC (H.264) no caso de vídeos de ação, que demandam até 3 Mbps.

É possível notar também que as abordagens **B** e **C** apresentam um comportamento mais regular do que a abordagem **A** para quantidades de clientes menores do que o NCP configurado para o *kernel*.

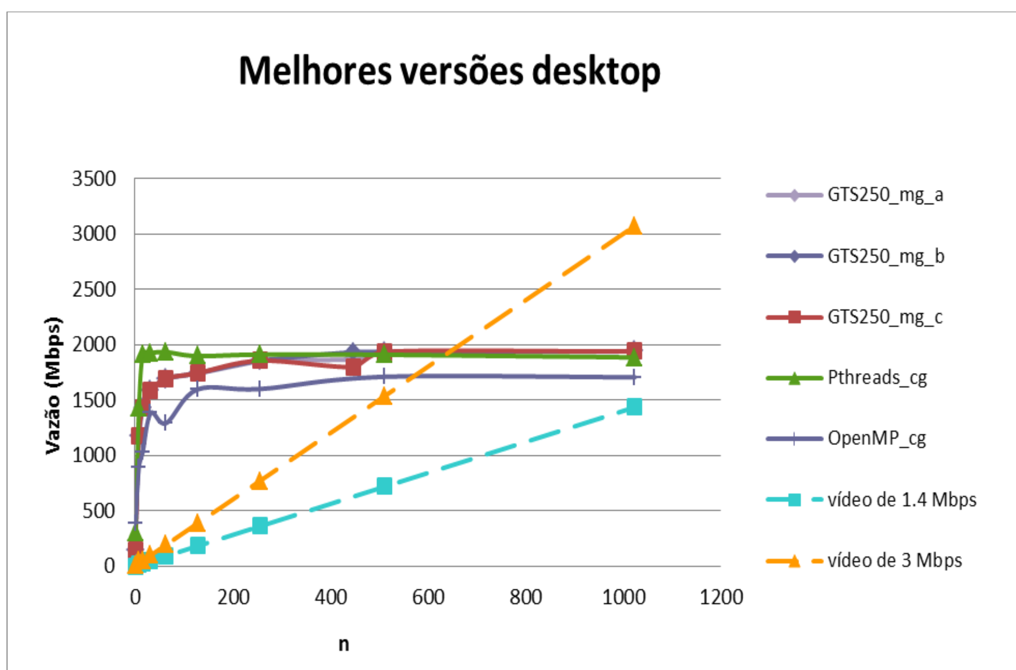
Outro cenário possível é quando há um *n* não múltiplo do NCP configurado (por exemplo *n* igual a 448), nesse caso a abordagem C leva vantagem em relação a abordagem B (em torno de 1,3%) pois os clientes serão atendidos concorrentemente em apenas uma chamada de *kernel*, ao passo que a abordagem B irá atender essa quantidade com mais chamadas de *kernel* com quantidades menores de clientes em concorrência, o que implica em mais transferências de memória entre o *host* e a GPU.

Figura 19 – Comparativo das melhores versões (servidor)



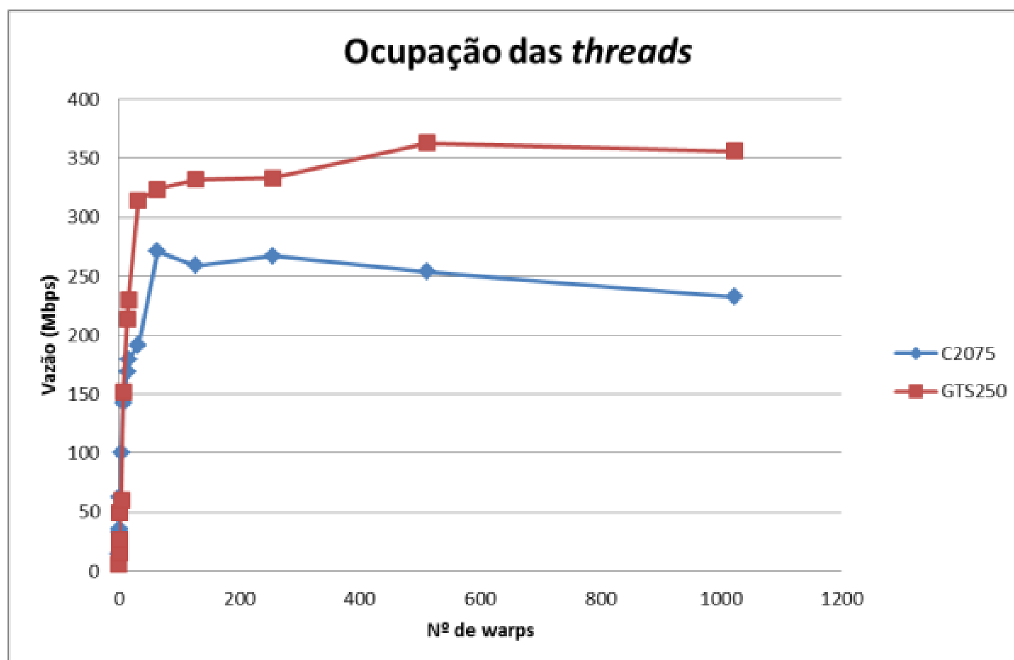
Fonte: Gomes (2012c).

Figura 20 – Comparativo das melhores versões (*desktop*)



Fonte: Gomes (2012c).

Figura 21 – Efeito da relação entre *warps* e núcleos da GPU



Fonte: Gomes (2012c).

Nota-se que a abordagem **B** na GTS 250 leva vantagem em relação a abordagem **C** (em torno de 6,9% para *n* igual a 448), devido ao fato de que o *kernel* que irá atender os clientes remanescentes tende a ter um NCP maior do que oito, gerando um *overhead* por exceder a capacidade de atendimento de clientes em paralelo a cada chamada desse *kernel*.

Analisando a Figura 20 tendo como referência a Figura 19, é perceptível que a GPU com a arquitetura Fermi (C 2075) fornece uma vazão significativamente maior que a vazão fornecida pela GTS 250 a partir de quantidades de clientes maiores do que oito, pois os *warps* irão trabalhar melhor sobre a mesma entrada (no caso da versão *mix-grained*) e com isso se beneficiar da cache da GPU e do modelo SIMD (*Single Instruction Multiple Data*). Além disso, nota-se que ao contrário do que ocorre com a C2075, só seria possível atender 1024 clientes paralelos no caso de vídeos com baixa movimentação (*Talking Head*). Adicionalmente, a C2075 além de possuir mais CUDA *cores* que a GTS 250 (448 contra 128 respectivamente), possibilitando atender mais clientes em paralelo com a versão *mix-grained*, também possui o *Gigathread Scheduler* que é responsável por escalonar os *warps* até 10x mais rápido com o escalonador das GPUs da série GT200.

A Figura 21 fornece as informações que caracterizam a ocupação das *threads* e o desempenho fornecido para cifrar um vídeo inteiro para um cliente. Quando

foram realizados os experimentos no servidor, notou-se que os resultados gerados pelas emulações com Pthreads e OpenMP foram semelhantes aos resultados obtidos em análise prévia (GOMES, 2012a). Devido a isso, os resultados do *desktop* para as versões *CPU-only* não são apresentados nos gráficos para fins comparativos.

Como previsto em (GOMES, 2012a), a vazão máxima fornecida por uma GPU Fermi se mostrou superior à da GTS 250 em aproximadamente 50%. No escalonamento da Fermi é possível ter dois *warps* executando no mesmo *Stream Multiprocessor,* sendo que cada um desses *warps* vai ser executado sobre 16 dos 32 CUDA *cores* disponíveis por SM. Em cada SM são intercaladas as instruções de ambos *warps* e executadas sobre os respectivos 16 CUDA *cores*. Esse detalhe da arquitetura explica porque é necessário utilizar grandes quantidades de *threads* cooperativas (*x*) nas versões que fazem uso da GPU para melhor explorar a sua capacidade de processamento paralelo. Quando a condição *x* < *Nº_de_CUDA_cores* é verdadeira, o desempenho nas GPUs é drasticamente reduzido. Quando *x* supera em grande amplitude *Nº_de_CUDA_cores*, o desempenho fornecido tende a ser superior. Seriam necessárias mais *threads* para a vazão começar a ser maior na C2075. No caso da GTS250 o número de *threads* necessárias para tirar melhor proveito da arquitetura da GPU é alcançado com uma menor quantidade de *threads*, pois tem menos CUDA *cores* do que a C2075 (Figura 21). E no caso onde somente um cliente é atendido por um número maior de *threads*, não há o aproveitamento do mesmo conjunto de dados, uma vez que os *warps* vão precisar acessar dados de entrada diferentes na memória global da GPU. Por isso nem sempre há uma melhora significativa quando o número de *warps* é incrementado; apenas quando são atendidos grandes números de clientes em paralelo é que a C2075 se beneficia do aumento da quantidade de *warps,* como pode ser observado nas Figuras 20 e 21.

Cabe destacar também que para quantidades de clientes maiores que 1024 e vídeos de 3 Mbps somente as versões que fazem uso da C2075 são capazes de atender a demanda da aplicação.

4.5.2 Análise do consumo energético

Para complementar a análise do custo-benefício dos módulos de recriptografia implementados, e determinar qual deles irá ser a solução mais

eficiente, foi feito uma análise do consumo energético de cada cenário experimentado e avaliado na métrica de desempenho, fazendo uso de um simulador de GPU, o GPGPU-Sim (BAKHOD, 2009) na versão 3.2.1.

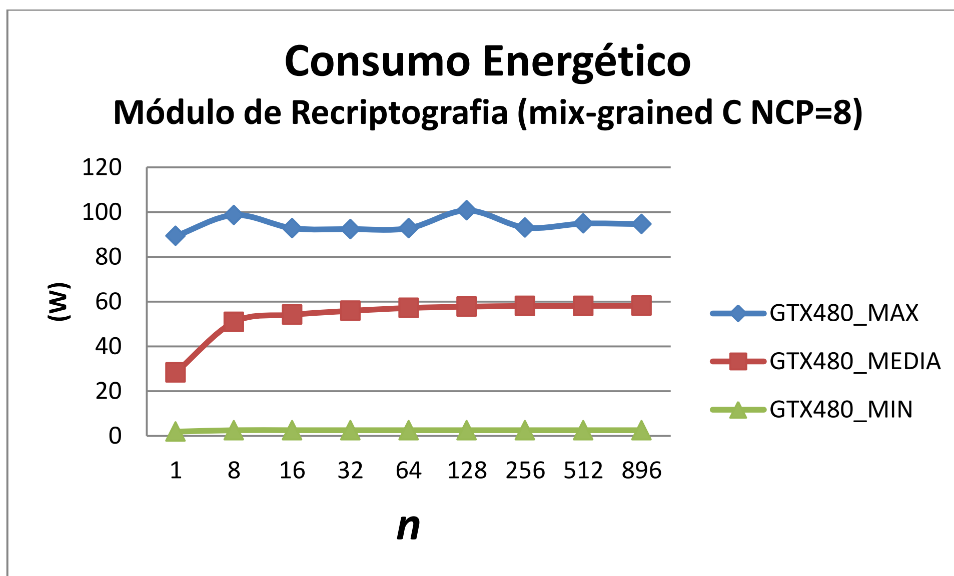
O GPGPU-Sim além de poder ser configurado em diversos aspectos arquiteturais da GPU (como tamanho da *cache*, de memória de constantes, memória compartilhada, número de SMs, etc), é capaz de fornecer estatísticas de acertos e erros no acesso aos níveis de memória da GPU. Esses recursos disponíveis no simulador permitem depurar uma aplicação GPGPU com um nível de detalhamento satisfatório. Essa versão do GPGPU-Sim também vem com um analisador de consumo energético GPUWattch (JINGWEN, 2013) na versão 1.0.

O GPUWattch é capaz de fornecer estatísticas do consumo energético em níveis bem pontuais da arquitetura da GPU como registradores, unidades de *fetch/decode,* unidades lógicas e aritméticas, etc. Para este trabalho não foi pertinente neste momento avaliar detalhadamente o consumo energético, porém foram considerados os dados referentes aos consumos mínimo, médio e máximo de cada cenário testado.

Uma vez que um simulador de uma arquitetura complexa, como é o caso de uma GPU, irá levar um tempo de execução relativamente alto (cerca de dez minutos, podendo chegar até cerca de três horas, dependendo do cenário testado), foram configurados *scripts* para iniciar os testes em todos os cenários (com GPU) testados para a métrica de desempenho (de 1 a 896 clientes, variando níveis de paralelismo e arquiteturas utilizadas, conforme descrito na seção 4.5.1). Esses *scripts* geraram *logs* com as principais informações pertinentes a análise da aplicação (módulo de recriptografia) como os *hits* e *misses* nos níveis de memória da GPU, consumos médio, mínimo e máximo de cada cenário. Após a geração dos resultados do GPGPU-Sim, foi montada uma planilha eletrônica para análise dessas informações, em contraste com os resultados de desempenho.

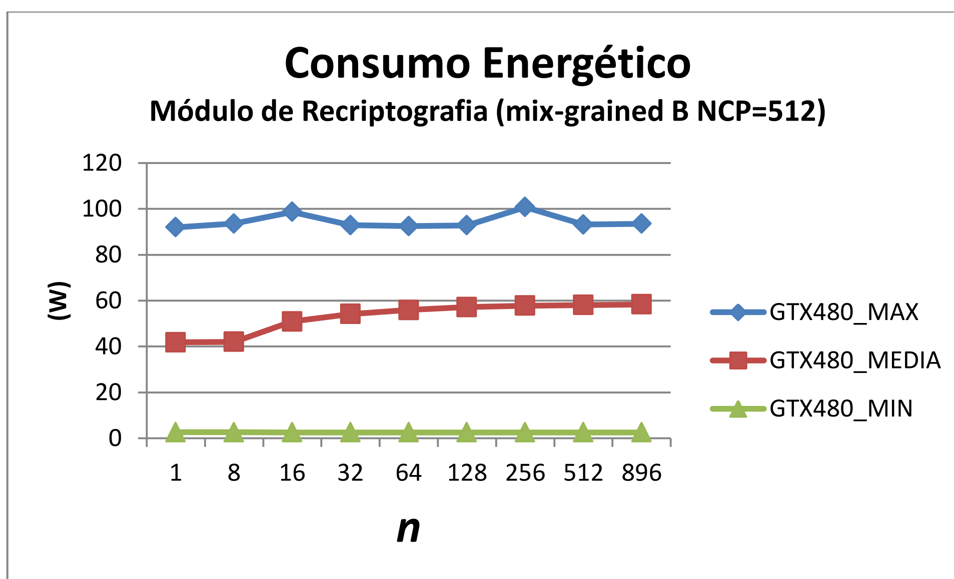
Os resultados preliminares para uma GPU da arquitetura Fermi (GTX 480) podem ser vistos na Figuras 22 e 23. Para a abordagem **C**, utilizando uma GPU Fermi, é possível notar que o consumo médio se mantém estável após 64 clientes, logo, também é vantagem, em termos de consumo energético, acrescentar mais clientes para serem atendidos por uma mesma execução do módulo de recriptografia (*kernel*) implementado com uma versão *mix-grained* adaptativa.

Figura 22 - Consumo Energético GTX 480 (MG_C_NCP=8)



Fonte: Gomes (2013).

Figura 23 - Consumo Energético GTX 480 (MG_B_NCP=512)



Fonte: Gomes (2013).

4.6 Restrições dos experimentos

Os resultados coletados a partir dos experimentos com as GPUs disponíveis, apesar de relevantes, não permitem grande generalizações por serem significativamente influenciados pelas características arquiteturais do hardware empregado. Ou seja, apesar de representarem tendências, são válidos apenas para

os ambientes experimentais nos quais os resultados foram coletados.

Para generalizar as análises, seria possível trabalhar com um universo mais variado de modelos de GPUs, permitindo coletar uma maior gama de resultados que fossem representativos de diferentes famílias da fabricante (NVIDIA) do hardware compatível com a API (CUDA) usada nas implementações do módulo de recriptografia. Essa opção demandaria grande investimento em hardware para viabilização.

Alternativamente, seria significativamente menos custo financeiramente e ainda mais flexível construir um modelo analítico que permitisse, mesmo que de forma aproximada, prever o desempenho obtido por um módulo de recriptografia aplicado à distribuição de vídeo quando empregado um hardware com determinadas características arquiteturais.

5 MODELO MATEMÁTICO (ANALÍTICO)

Conforme mencionado no Capítulo 1, para resolver o problema de DRM, os trabalhos anteriores propuseram o uso de GPUs para implementação do módulo de recriptografia de vídeo aplicada a um sistema de distribuição de vídeo sob demanda, tomando por base os conhecimentos básicos apresentados no Capítulo 2, e correlacionando-os com o estado da arte apresentado no Capítulo 3, a luz das contribuições trazidas pelo Capítulo 4, o presente capítulo propõe e descreve em detalhes um modelo matemático concebido para estimar o desempenho aproximado fornecido por uma GPU aplicada no módulo de recriptografia de vídeo de um sistema de distribuição.

O modelo analítico proposto se origina de uma adaptação do modelo de Barlas (2011), modificado de acordo com as características do módulo de recriptografia. Adaptação esta que culminou no desenvolvimento de uma planilha inicial para avaliar esse modelo no contexto da aplicação.

Primeiramente foi feita a implementação (formatação de uma planilha eletrônica no Excel) das duas equações principais para estimação do tempo de execução em cada nodo *i:*

$$T_i^{(PB)} = l_i \left(part_{0,i} L + part_{M-1,i} L + k\right) + 2a_i + p_i L \sum_{j=0}^{M-1} part_{j,i} \quad (3)$$

$$T_i^{(CB)} = part_i \frac{L}{M} (l_i (M+1) + p_i) + (M+1) a_i + l_i k \quad (4)$$

Onde:

PB: *Computation Bound*

CB: *Communication Bound*

l_i: inverso da vazão

p_i: inverso da velocidade computacional de Pi

a_i: latência

L: total de dados a serem cifrados

k: tamanho da chave de criptografia

M: número de *installments*

$part_i$: porcentagem do total do dado L processado pelo nodo

Logo após essa planilha foi preenchida com as variáveis de acordo com o ambiente experimental através da tabela abaixo foram preenchidos os valores das

variáveis:

Tabela 3 – Especificações das GPUs e mapeamento para o modelo

	GTS 250	C2075	GTX 580	*Mapping*
CUDA Cores	128	448	512	M
Graphics Clock	0.738 GHz		0.772 GHz	
Processor Clock	1.836 GHz	1.150 GHz	1.544 GHz	$p_i = \frac{1}{clock}$
Number of Multiprocessors	16	14	16	
CUDA Cores/ Multiprocessor	8	32	32	
GPU AES for VoD processing time (512 clients) in: ~4,8MB	11.096 s	6.379 s	4.227 s	
Memory Clock	1.1 GHz	1.5 GHz	2.0 GHz	$a_i = \frac{400}{clock}$
Standard Memory Config	0.5 GB DDR3	6 GB GDDR5	1.5 GB GDDR5	
Memory Interface Width	256-*bit*	384-*bit*	384-*bit*	
Memory Bandwidth	70.4 GB/s	144 GB/s	192.4 GB/s	l_i

Fonte: Gomes (2013).

Supondo que a latência seja o atraso no acesso a memoria global e que no pior caso sempre gaste 400 ciclos (valor *default* para acesso a memória global na GTS 250), então utiliza-se o *clock* da memória para calcular essa latência.

No modelo de (BARLAS, 2011) a criptografia é feita utilizando apenas uma chave para todo o dado de entrada (L) e $L_{in} = L_{out}$. No módulo de recriptografia, existe um cenário com ***n*** clientes para um mesmo vídeo, sendo que cada cliente tem a sua própria chave de criptografia, logo $L_{in} < L_{out}$, onde ($L_{out} = \boldsymbol{n}\, L_{in}$).

Levando em consideração a restrição anterior, num primeiro momento, foi feita uma abstração onde é utilizado o total de dados processados, que é o próprio L_{out}. Além disso, cabe mencionar que o tamanho da chave de criptografia (***k***) foi mantido em 256 *bits.*

Para as parcelas de computação (***part$_i$***), supõe-se que cada um dos

elementos de processamento (*CUDA cores*) contribui igualitariamente para o processamento total dos dados, logo $\boldsymbol{part_i} = 1/M$.

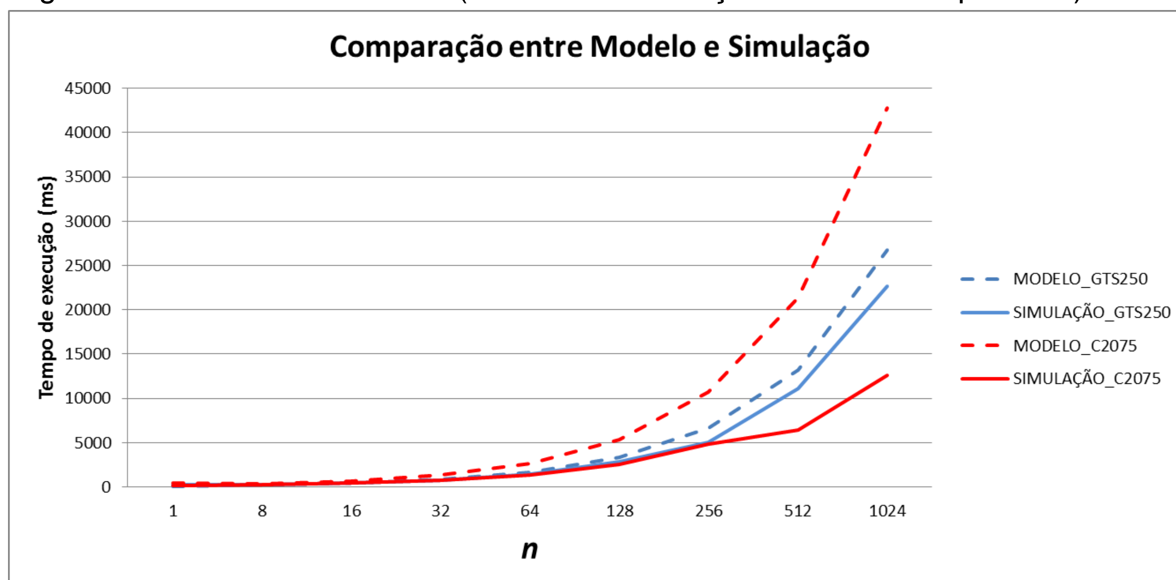
Naturalmente, foi necessário fazer conversões nas variáveis de entrada para que todas estivessem na mesma unidade. Em um primeiro momento o fator multiplicativo, que ajusta a curva do desempenho para cada nodo (CUDA *core*) da GTS 250, foi estipulado experimentalmente (de modo manual) com o valor de 0,001. Para diminuir a diferença dos resultados para C2075, em um primeiro momento, um fator multiplicativo (*z*) foi inserido na equação. Esse fator multiplicativo foi a divisão do número de CUDA *cores* da C2075 pelo número da GTS 250:

$$z = \frac{CUDA\ cores\ C2075}{CUDA\ cores\ GTS250} * 0{,}001 = \frac{448}{128} * 0{,}001 = 3{,}5 * 10^{-3} \quad (5)$$

Os resultados fornecidos pela primeira abordagem se mostraram compatíveis com os resultados experimentais (Simulação) realizados anteriormente e discutidos no Capítulo 4. No caso da GPU GTS 250, o modelo de computação *processing bound* foi o que demonstrou um resultado mais próximo do valor real de simulação (em media 12,5% de diferença), conforme é possível ver no gráfico (Figura 24).

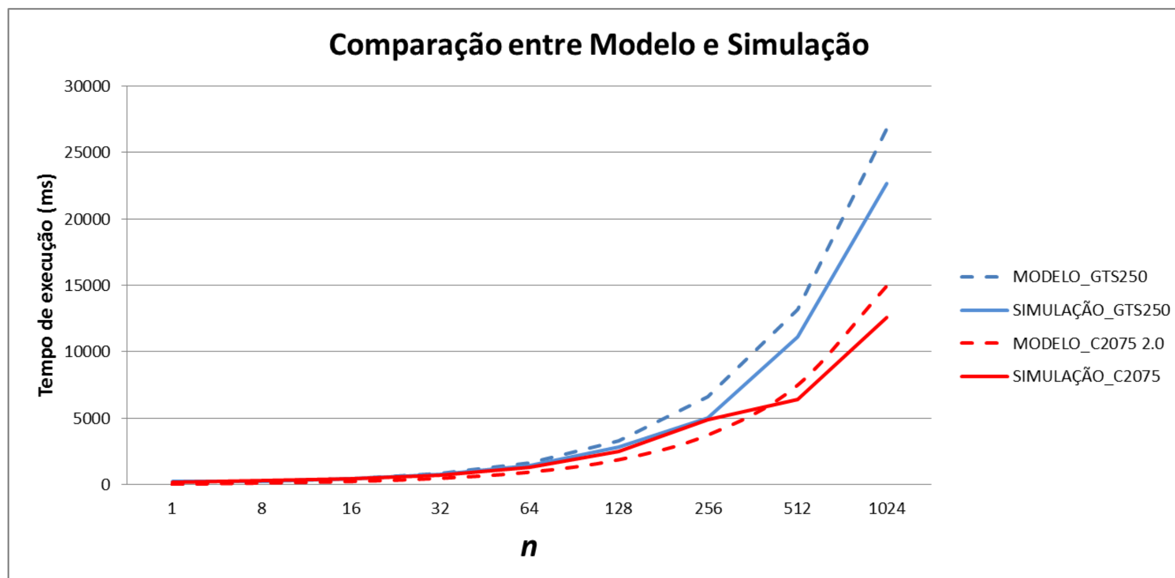
Já no caso da C2075, resultado gerado pelo modelo, sem adaptações, diverge bastante utilizando somente a analogia direta do modelo matemático sem as devidas adequações referentes à arquitetura Fermi, como pode ser observado na Figura 25, tomando por base a Figura 24.

Figura 24 – Resultados iniciais (C2075 sem correção do fator multiplicativo)



Fonte: Gomes (2013).

Figura 25 – Resultados iniciais (C2075 com correção do fator multiplicativo)



Fonte: Gomes (2013).

Posteriormente, após testes mais elaborados, chegou-se a um valor mais preciso para o fator multiplicativo (0,00913), que melhor ajusta a curva do desempenho para cada nodo (CUDA *core*) da GTS 250.

Quando o simulador de GPU foi utilizado para determinar o consumo energético da aplicação para cada cenário experimentado anteriormente, conforme relatado na seção 4.5.2, foi observado que havia estatísticas para os fatores pertinentes ao modelo matemático. Para o valor estipulado anteriormente, os fatores que influenciam são taxa de *hits* e *misses* de leitura e escrita nos níveis de *cache* L2, *cache* de constantes, *cache* de dados e w*arp scheduler* que influencia nas taxas de acesso a *cache* de instruções dos SMs. Embora se soubesse que esses fatores influenciam no modelo matemático, não se sabia como eles interagiam com o modelo proposto. Não só para descobrir como esses fatores se integram com o modelo, mas também para realizar um refinamento no modelo proposto inicialmente, foram supostos novos fatores multiplicativos de acordo com os cenários (mais especificamente as abordagens das versões *mix-grained* adaptativas, onde o NCP varia).

Levando em consideração as variáveis NCP, número de CUDA cores, arquiteturas diferentes (famílias GT200 e Fermi), foram feitos testes manuais na planilha, alterando e multiplicando as estatísticas (fatores) determinadas pelo simulador de GPU. Após vários testes, notou-se que tomando por base a razão

(média) entre *hits* e *misses* das escritas na *cache* L2 e descontando as demais razões de *hits* e *misses* (nas *caches* de leitura na *cache* L2, na de constantes e na de instruções), um novo fator multiplicativo (**y**), em módulo, pode ser aplicado no total de dados processados (***part****i*). Além disso, para completar a generalização do modelo, o fator multiplicativo ***f*** ajusta os resultados gerados para a GPU a ser testada, uma vez que as estatísticas obtidas pelo GPGPU-Sim são da GTX 480. Essa descoberta implica em uma nova equação mais precisa e genérica (conforme será apresentado logo a frente) para o modelo matemático proposto:

$$T_i^{(PB)} = f\left[l_i\left(y(part_{0,i}L) + y(part_{M-1,i}L) + k\right) + 2a_i + p_i L y \sum_{j=0}^{M-1} part_{j,i}\right] \quad (6)$$

$$y = \left|\frac{\Delta L2_WH}{\Delta L2_WM} - \frac{\Delta L2_RH}{\Delta L2_RM} - \frac{\Delta CC_H}{\Delta CC_M} - \frac{\Delta DC_RH}{\Delta DC_RM}\right| \quad (7)$$

$$f = \frac{CUDA\ cores\ GPU}{CUDA\ cores\ Fermi} \quad (8)$$

Onde,

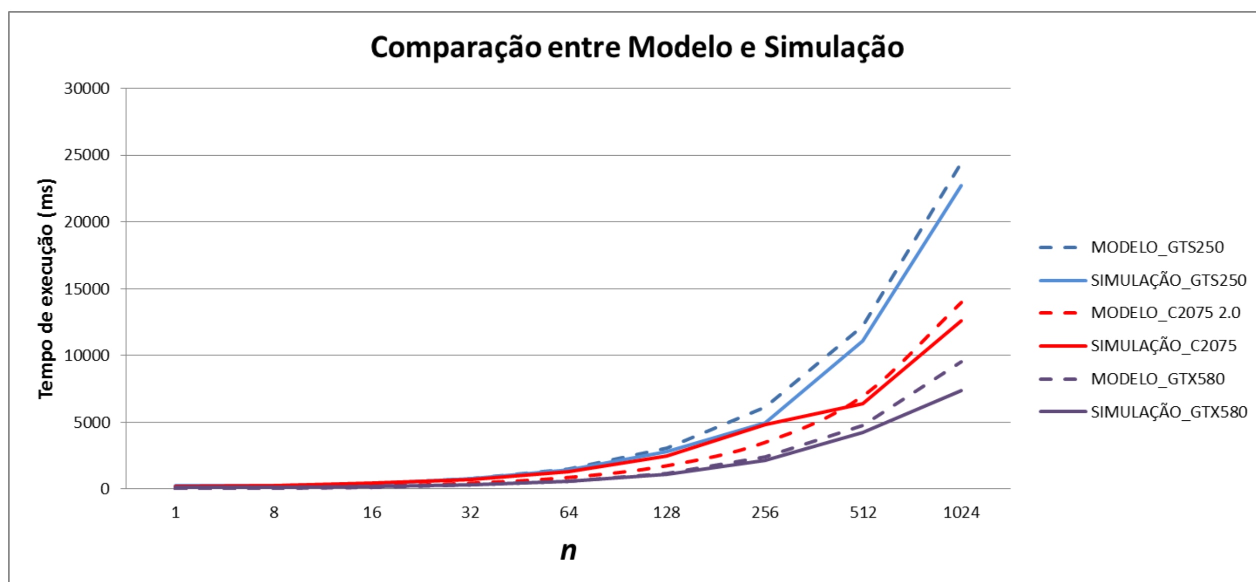
$\Delta \boldsymbol{L2_WH}$: número de *hits* de escrita na *cache* de nível L2 da GPU
$\Delta \boldsymbol{L2_WM}$: número de *misses* de escrita na *cache* de nível L2 da GPU
$\Delta \boldsymbol{L2_RH}$: número de *hits* de leitura na *cache* de nível L2 da GPU
$\Delta \boldsymbol{L2_RM}$: número de *misses* de leitura na *cache* de nível L2 da GPU
$\Delta \boldsymbol{CC_RH}$: número de *hits* de leitura na *cache* de constantes da GPU
$\Delta \boldsymbol{CC_RM}$: número de *misses* de leitura na *cache* de constantes da GPU
$\Delta \boldsymbol{DC_RH}$: número de *hits* de leitura na *cache* de dados da GPU
$\Delta \boldsymbol{DC_RM}$: número de *misses* de leitura na *cache* de dados da GPU

Para a GTS 250, o NCP foi parametrizado com oito (NCP=8), pois foi o *best-fit* (encontrado experimentalmente na análise de desempenho) para essa GPU. Levando isso em consideração, nos resultados fornecidos pelo GPUWattch, é possível notar que no cenário da *mix-grained* **A**, em média e módulo, diferença entre essas taxas equivale a 0,010080811, o qual é o valor próximo do fator multiplicativo estipulado (com diferença de 9,43%). Para completar o ajuste desse fator multiplicativo, outros fatores podem influenciar (como o erro de precisão do

próprio GPUWattch), pois como se trata de um valor médio e o comportamento da aplicação (criptografia) na GPU não é tão regular quanto o de uma CPU, conforme já demonstrado por Barlas (2011) .

Uma vez que o modelo foi melhorado e se tornou mais preciso, se fez necessária a validação desse modelo através da comparação dos resultados gerados pelo modelo para uma terceira GPU, a GTX 580, com os resultados experimentais gerados posteriormente. Ela também é da família Fermi, porém possui 512 núcleos CUDA e *clocks* de elementos de processamento e de memória superiores a C2075, conforme pode ser visto na Tabela 3. Os resultados para o modelo matemático (final) proposto podem ser visualizados na Figura 26.

Figura 26 - Comparação entre o modelo matemático e simulação



Fonte: Gomes (2013).

6 CONSIDERAÇÕES FINAIS

Este trabalho realiza a interface entre sistemas VoD e criptografia assistida por GPU, de tal forma que foi feita uma análise qualitativa e quantitativa de desempenho de implementações de uma solução paralelizada do cifrador AES baseadas em CUDA, Ptheads e OpenMP, testadas em diferentes sistemas computacionais com CPU *multicore* e GPU. O algoritmo de criptografia escolhido foi o AES no modo ECB (por apresentar forte potencial para exploração do paralelismo no modelo SIMD), porém, na etapa de revisão bibliográfica, foram considerados e estudados outros algoritmos como o CTR e o RC6 (que atualmente está em ascensão na área, devido a sua simplicidade). Após uma bateria de experimentos, foi possível verificar que as GPUs tendem a ser mais eficientes (em termos de desempenho) para esta aplicação, principalmente em cenários onde a capacidade computacional é mais demandada (que ocorre quando o *proxy* de vídeo atende uma grande quantidade de clientes concorrentes). Além disso, foi possível demonstrar a influência da estratégia de adaptação do paralelismo na vazão obtida com diferentes arquiteturas de GPUs em um servidor e um desktop. As abordagens adaptativas (B e C) superam em até 33,2% a abordagem original (a), sendo a média 23,8% para o servidor, enquanto que para o *desktop* não há influência de adaptação.

Os resultados analíticos iniciais, gerados através do modelo matemático proposto, se mostraram compatíveis com os valores fornecidos pelas simulações nos ambientes experimentais. Com a primeira abordagem os tempos de execução estimados para GTS 250 e para C2075 diferem aproximadamente, em média, de %12,5 e %44,5 respectivamente. Já com o ajuste proposto pelo fator multiplicativo z a diferença para a C2075 caiu para %27,1 aproximadamente.

Conforme mencionado no Capítulo 5, o uso do simulador GPUWattch foi importante para refinar o modelo matemático, uma vez que ele é capaz de fornecer estatísticas pertinentes do uso dos componentes das GPUs para as diferentes versões implementadas do módulo de recriptografia. Com esses resultados fornecidos foi possível aprimorar o modelo matemático de tal forma que a diferença entre o modelo e os resultados experimentais foi reduzida, em média, para 9,4% para GTS 250, 15% para a C2075 e 14,1% para a GTX 580. Esses resultados são satisfatórios para validar o modelo matemático, em perspectiva de generalização, uma vez que são próximos o suficiente de um tempo de execução para determinado

cenário com outra GPU, sem a necessidade direta de realizar experimentos nela antes de uma possível aquisição para determinado módulo de recriptografia de vídeo assistido por GPU.

Para completar as tarefas que o presente trabalho se propôs, foi realizada uma mensuração do consumo de energia para as implementações (para as versões *mix-grained* adaptativas que fazem uso de uma GPU) propostas no Capítulo 4, através do GPUWattch, que além dessas estatísticas pertinentes, que gerou os resultados para o consumo energético desses cenários experimentados ao longo do desenvolvimento deste trabalho. Para analisar e determinar a solução de melhor custo-benefício (contraste entre as métricas desempenho, consumo energético e preço) para implementação do módulo, levando em consideração todas as implementações, seriam necessários os dados referentes ao consumo energético para as versões *CPU-only* que fazem uso apenas de *multicores*.

REFERÊNCIAS

ALVES, J. M. O. ; PINHO, L. B. ; AMORIN, C. L. **Glove-Mix Integrando Peer-To-Peer e Distribuição Via Proxy em Serviços de Vídeo sob Demanda**. In: V Workshop de Redes Dinâmicas e Sistemas P2P, 2009.

BAKHOD, A. et al. **Analyzing CUDA Workloads Using a Detailed GPU Simulator**. In: Proceedings of the IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS), Boston, MA, EUA, v. 1, p. 163-174, 2009. April 26-28, 2009.

BARLAS, G. et al. **An Analytical Approach to the Design of Parallel Block Cipher Encryption/Decryption: A CPU/GPU Case Study**. In: 19th Euromicro International Conference on (PDP), v. 1, p. .247-251, 2011. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5739008&isnumber=5738968>. Acesso em: 30 abr. 2013.

BUTENHOF, D. R. **Programming with POSIX Threads**. Addison-Wesley, 1997.

DAEMEN, J., RIJMEN, V. **The Design of Rijndael**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2002. ISBN 3540425802.

GILL, P. et al. **Youtube traffic characterization: A view from the edge, imc**. In: Proc. of IMC. [S.l.: s.n.], 2007.

GOMES, L. A. S. ; NEVES, B. S. ; PINHO, L. B. **Proposta e Avaliação de Recriptografia Assistida por GPU Aplicada a Servidores Escaláveis de Distribuição de VoD**. In: 11a Escola Regional de Alto Desempenho (ERAD), Porto Alegre, RS, Brasil, v. 1, p 181-184, 2011. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erad-rs/2011/0017.pdf>. Acesso em: 30 abr. 2013

______. **CUDA x OpenMP x Pthreads: Implicações no custo total de uma solução de distribuição segura de vídeos**. In: 12a Escola Regional de Alto Desempenho (ERAD), Erechim, RS, Brasil, v. 1.p. 157-160, 2012. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erad-rs/2012/0039.pdf>. Acesso em: 30 abr. 2013

______. **Empirical Analysis of Multicore CPU and GPU-based Parallel Solutions to Sustain Throughput Needed by Scalable Proxy Servers for Protected Videos**. In: XIII Simpósio em Sistemas Computacionais (WSCAD-SSC), Petrópolis, RJ, Brasil, v. 1, p. 49-56, 2012. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/WSCAD-SSC.2012.37>. Acesso em: 30 abr. 2013

______. **Adaptive Parallel Approaches for GPU-based Reencryption Applied in Scalable Proxy Servers for Protected Video Distribution**. In: I Workshop em Modelos de Programação Paralela (MPP), Petrópolis, RJ, Brasil, v. 1, p 226-233,

2012. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/WSCAD-SSC.2012.40>. Acesso em: 30 abr. 2013

GRANADO, A. **Avaliação Experimental da Memória Cooperativa Colapsada para Sistemas de Vídeo sob Demanda.** Dissertação (Mestrado em Engenharia de Sistemas e Computação), COPPE/UFRJ, Rio de Janeiro, 2010.

HAIVISION. **MPEG-4 AVC (H.264) and Why, Only Now, It Can Save 60% of the Network Video Bandwidth and Storage Requirements**. 2011. Disponível em: <http://www.haivision.com>. Acesso em: 27 nov. 2011.

IEEE COMPUTER SOCIETY. **Accelerators and GPUs track**. São Paulo, Brasil, 2009. IEEE SBAC-PAD.

JINGWEN, L. et al. **GPUWattch: Enabling Energy Optmization in GPGPUs.** In: 40th International Symposium on Computer Architecture (ISCA), Tel-Aviv, Israel, 2013.

MANAVSKI, S. **A Cuda compatible gpu as an ecient hardware accelerator for aes cryptography**. 2007 IEEE International Conference on Signal Processing and Communications, IEEE, v. 9 Suppl 2, n. November, p. 65-68, 2007. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4728256>. Acesso em: 3 nov. 2011.

MARTINS, T. X. **Desenvolvimento e Comparação de Diferentes Arquiteturas em Hardware (VHDL) para Criptografia AES em FPGA**. Bagé: Engenharia de Computação, 2013.

MURUGESAN, S. **Harnessing green it: Principles and practices**. IT Professional, IEEE Computer Society, Los Alamitos, CA, USA, v. 10, p. 24-33, 2008. ISSN 1520-9202.

NVIDIA. **O que é CUDA?** 2011. Disponível em: <http://www.nvidia.com.br/object/what_is_cuda_new_br.html>. Acesso em: 3 nov. 2011.

______. **NVIDIA Fermi Compute Architecture Whitepaper**. 2011. Disponível em: <http://www.nvidia.com/content/PDF/fermi_white_papers>. Acesso em: 28 nov. 2011.

______. **CUDA C Programming Guide.** 2012. Disponível em: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Acesso em: 30 abr. 2012.

OPENMP. **OpenMPAPI**. 2013. Disponível em: < http://openmp.org/wp/>. Acesso em: 30 abr. 2013.

OPENSSL. **OpenSSL Library**. 2011. Disponível em: <http://www.openssl.org/>. Acesso em: 3 nov. 2011.

SANTANA, H. **Qualidade de serviço (QoS) em redes IP: princípios básicos, parâmetros e mecanismos**. 2006. Disponível em: <http://professores.unisanta.br/santana/downloads%5CTelecom%5CCom_Digitais%5CAulas%202o.%20Bimestre%5CTexto%20QoS_IP_Itelcon.pdf >. Acesso em: 30 abr. 2013.

SESHADRINATHAN, M., DEMPSKI, K. L. **Implementation of Advanced Encryption Standard for Encryption and Decryption of Images and Text on a GPU.** IEEE Computer Vision and Pattern Recognition Workshops (CVPRW), Anchorage, AK, USA, v. 1, p. 1-6,2008.

SUBRAMANYA, S. R.; YI, B.K. Digital rights management. **Potentials,** IEEE, v. 25.Suppl2.p. 31-34, 2006. Disponível em <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1649008&isnumber=34572>. Acesso em: 30 abr. 2013.

TANENBAUM, Andrew. **Sistemas Operacionais Modernos. 3. ed.** São Paulo: Prentice Hall, 2010.

TECHNOLOGY, N. I. O. S. A. **Advanced Encryption Standard (AES) (FIPS PUB 197)**. Federal Information Processing Standards Publication 2001.

YEUNG, S. F., LUI, J. C. S., YAU, D. K. Y. **A Multi-key Secure Multimedia Proxy Using Asymmetric Reversible Parametric Sequences: Theory, Design, and Implementation.** IEEE Transactions on Multimedia, v. 7, 2005.