UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS TRINDADE

Leandro Perin de Oliveira

USO DE COMPUTAÇÃO PARALELA PARA ACELERAR A CRIPTO-COMPRESSÃO DE DADOS

FLORIANÓPOLIS
2017

LEANDRO PERIN DE OLIVEIRA

USO DE COMPUTAÇÃO PARALELA PARA ACELERAR A CRIPTO-COMPRESSÃO DE DADOS

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina, como requisito necessário para obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Prof. Dr. Márcio Bastos Castro

Florianópolis
2017

RESUMO

Algoritmos de compressão e cifragem de dados eficientes são bastante necessários no atual cenário da computação. Serviços executados na web precisam processar grandes quantias de dados, como imagens e vídeos, e entregar rapidamente o resultado aos usuários. Neste trabalho, proponho uma solução paralela para um algoritmo de criptografia de dados, visando minimizar o tempo de execução do mesmo.

Palavras-chave: Criptografia. Análise de desempenho. Computação de alto desempenho. Sistemas paralelos.

Lista de ilustrações

Lista de tabelas

Siglas

AES *Advanced Encryption Standard*. 23

ALU *Arithmetic Logic Unit*. 18

API *Application Programming Interface*. 15, 19, 20, 35

BMP *Bitmap*. 29

CPU *Central Processing Unit*. 15, 16, 18, 21, 36

CUDA *Compute Unified Device Architecture*. 20, 21, 35, 36, 37, 41

DCT *Discrete Cosine Transform*. 29, 30

DES *Data Encryption Standard*. 23

DH *Diffie-Hellman Key Exchange*. 24

ELGAMAL *ElGamal Encryption System*. 24

GIF *Graphics Interchange Format*. 11, 27

GMPR *Geometric Modelling and Pattern Recognition Group*. 11, 12, 29, 30, 31, 35, 37, 41

GPGPU *General Purpose Graphics Processing Unit*. 18

GPU *Graphics Processing Unit*. 11, 12, 20, 21, 36, 37

JPEG *Joint Photographic Experts Group*. 11, 27, 29

JPEG2000 *Joint Photographic Experts Group 2000*. 26, 29

LUT *Look Up Table*. 29

LZ *Lempel Ziv*. 27

LZW *Lempel Ziv Wech*. 27

MD5 *Message Digest*. 25

MIMD *Multiple Instruction, Multiple Data*. 15, 16

MISD *Multiple Instruction, Single Data*. 15, 16

MIT *Massachusetts Institute of Technology*. 24, 25

MPI *Message Passing Interface*. 15, 19, 20, 21, 35, 36, 37, 41

NORMA *Non-Remote Memory Access*. 18

NSA *National Security Agency*. 25

NUMA *Non-Uniform Memory Access*. 17

OPENCL *Open Computing Language*. 36, 37, 41

OpenMP *Open Multi-Processing*. 19, 21, 35, 36, 37, 41

PNG *Portable Network Graphics*. 11, 26, 27

RLE *Run Lenght Encoding*. 27

RSA *Rivest-Shamir-Adleman*. 24

SHA *Secure Hash Algorithm*. 25

SIMD *Single Instruction, Multiple Data*. 15, 18

SIMT *Single Instruction, Multiple Threads*. 18

SISD *Single Instruction, Single Data*. 15

TIFF *Tagged Image File Format*. 11, 26, 27

UMA *Uniform Memory Access*. 17

Sumário

1 Introdução

Serviços como o YouTube, Instagram, Facebook, dentre outros, possuem uma enorme quantidade de dados armazenados, incluindo texto, imagens e vídeos. Com o aumento no uso de serviços como esses, maior tráfego de dados através da rede e necessidade de armazenamento cada vez maiores, são requeridos métodos mais eficientes para compressão de imagens e vídeos, com alta qualidade de reconstrução e redução na quantidade de armazenamento necessária para os dados [Rodrigues e Siddeq 2016].

Atualmente existem diversos algoritmos que comprimem imagens. Alguns garantem fidelidade máxima, chamados de compressão sem perdas, que é o caso dos formatos *Portable Network Graphics* (PNG) e *Tagged Image File Format* (TIFF), enquanto outros acabam perdendo parte da imagem original, chamados de compressão com perdas, que é o caso dos formatos *Joint Photographic Experts Group* (JPEG) e *Graphics Interchange Format* (GIF). Vale mencionar que esses algoritmos não fazem uso de criptografia, eles apenas comprimem as imagens [Salomon 2007].

Portanto, a disponibilidade de um algoritmo de compressão e cifragem de dados mais eficiente, que consiga reduzir mais o tamanho dos arquivos, comprimir e criptografar de forma rápida, e ainda assim mantendo um bom desempenho na recuperação dos dados é algo que melhoraria bastante o uso de serviços com alto tráfego de informações. Tal algoritmo tornaria os serviços em nuvem mais populares, afinal deixaria os mesmos mais rápidos e seguros, encorajando o desenvolvimento de cada vez mais aplicativos que fazem uso de muitos dados [Stallings 2014].

Um algoritmo de cripto-compressão de dados, denominado *Geometric Modelling and Pattern Recognition Group* (GMPR) [Rodrigues e Siddeq 2016], foi proposto recentemente por pesquisadores da *Sheffield Hallam University*. O GMPR faz uso de técnicas de cifragem para garantir a segurança dos dados ao mesmo tempo que busca reduzir o tamanho dos arquivos.

Embora o algoritmo de cripto-compressão em questão funcione corretamente, ele ainda demora um tempo considerável para processar conteúdos grandes, como imagens de alta resolução ou textos com milhares de linhas, o que torna inviável o uso do mesmo em aplicações que exigem resposta rápida aos usuários.

1.1 Objetivos

Com base no exposto, são apresentados a seguir o objetivo geral e os objetivos específicos do presente projeto.

1.1.1 Objetivo Geral

O objetivo geral deste TCC é propor e implementar uma versão paralela eficiente do algoritmo de cripto-compressão GMPR desenvolvido pela *Sheffield Hallam University* para processadores paralelos tais como *multicores* e *Graphics Processing Units* (GPUs). A solução proposta permitirá reduzir significativamente o tempo de execução do algoritmo de cripto-compressão GMPR.

1.1.2 Objetivos Específicos

Os objetivos específicos são listados a seguir:

- Produzir um código sequencial limpo e organizado do algoritmo GMPR em C++ com base na implementação existente em Matlab [Rodrigues e Siddeq 2016];
- Propor uma solução paralela para o algoritmo para arquiteturas *multicore*;
- Propor uma solução paralela do algoritmo para GPUs;
- Realizar experimentos com o intuito de medir o desempenho das soluções propostas em diferentes processadores *multicore* e GPUs.

1.2 Justificativa

Este trabalho se insere em uma colaboração inicial entre o Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD) da UFSC e a *Sheffield Hallam University* (Reino Unido), proponente e desenvolvedora do algoritmo GMPR. A necessidade de paralelizar o código veio de seus problemas de desempenho, o que inviabiliza o uso em aplicações reais. Esse algoritmo funcionando de forma eficiente e entregando rapidamente o resultado aos usuários é algo que iria reduzir problemas de armazenamento e segurança dos dados, pois poderia cifrar e comprimir os mesmos, recuperando-os corretamente no futuro, sem se tornar um gargalo na aplicação. Este trabalho poderá, também, ser integrado facilmente em projetos futuros que tenham restrições de infraestrutura, necessidade de maior velocidade de processamento ou maior segurança das informações, dentre outras razões.

Este TCC permitirá estudar as melhores técnicas de computação paralela que possam auxiliar no desenvolvimento do trabalho proposto. Tais técnicas serão utilizadas no projeto, implementadas e testadas para que se possa escolher uma ou mais que se mostrem adequadas para o desenvolvimento do trabalho.

1.3 Organização do Texto

O texto deste trabalho será organizado da seguinte forma. A Seção 2 apresenta e descreve a fundamentação teórica deste trabalho. A Seção 3 apresenta o algoritmo de cripto-compressão GMPR, incluindo suas principais ideias e pseudo-código. Então, a Seção 4 apresenta algumas ideias e possibilidades de paralelização do algoritmo GMPR para arquiteturas *multicore* e GPUs. A Seção 5 apresenta as atividades e o que será feito em cada uma delas. Por fim, as principais observações a serem feitas a respeito do trabalho a ser desenvolvido e possíveis melhorias e usos deste trabalho em projetos futuros são descritas na Seção 6.

2 Fundamentação Teórica

2.1 Computação Paralela

Há muito tempo o mercado de tecnologia vem buscando cada vez mais velocidade de processamento. Várias áreas demandam muito poder computacional para executar suas tarefas, desde o sequenciamento de DNA até simulações do universo, e para isso necessitam cada vez mais de um maior desempenho.

Antigamente, o aumento de desempenho estava diretamente atrelado ao aumento da frequência dos processadores. Porém, este aumento atingiu um limite físico para a velocidade do relógio. Quando mais rápida a frequência de relógio, menor deve ser o processador, pois os sinais elétricos devem ir e voltar dentro do mesmo ciclo de relógio. A solução para isso poderia ser a construção de uma *Central Processing Unit* (CPU) cada vez menor, porém isso causa problemas de aquecimento, afinal uma maior frequência de relógio gera mais calor. Com a redução do tamanho da CPU torna-se mais complicada a dissipação de calor da mesma.

Executar várias tarefas em paralelo, com várias CPUs trabalhando em conjunto, acabou sendo a forma encontrada para aumentar o poder computacional das máquinas. Hoje em dia já é possível encontrar arquiteturas paralelas com milhares de CPUs. Essas arquiteturas podem ser classificadas em dois grandes grupos: multiprocessadores e multicomputadores. No primeiro grupo (multiprocessadores) encontram-se as arquiteturas compostas por diversas CPUs interligadas através de um barramento ou similar, permitindo assim um compartilhamento da memória principal entre todas as CPUs. Por outro lado, no segundo grupo (multicomputadores), a memória principal é distribuída. A programação paralela nessas arquiteturas é feita através do uso de linguagens de programação, como o Erlang, e *Application Programming Interfaces* (APIs), como o *Message Passing Interface* (MPI), desenvolvidos especialmente para computação paralela [Tanenbaum e Bos 2015].

2.1.1 Arquiteturas Paralelas

Arquiteturas paralelas podem ser classificadas segundo seu fluxo de instruções e fluxo de dados utilizando a Taxonomia de Flynn [Flynn 1972]. A Tabela 1 mostra as quatro classes possíveis de arquiteturas segundo esta classificação.

SISD: Um único fluxo de instruções trabalha em um único fluxo de dados. É a representação clássica de uma arquitetura sequencial. A Figura 1 ilustra essa classe.

SIMD: Uma única instrução é responsável pelo processamento de vários dados.

	Único	Múltiplo
Único	*Single Instruction, Single Data* (SISD)	*Single Instruction, Multiple Data* (SIMD)
Múltiplo	*Multiple Instruction, Single Data* (MISD)	*Multiple Instruction, Multiple Data* (MIMD)

Tabela 1 – Taxonomia de Flynn

Figura 1 – Classe SISD da Taxonomia de Flynn

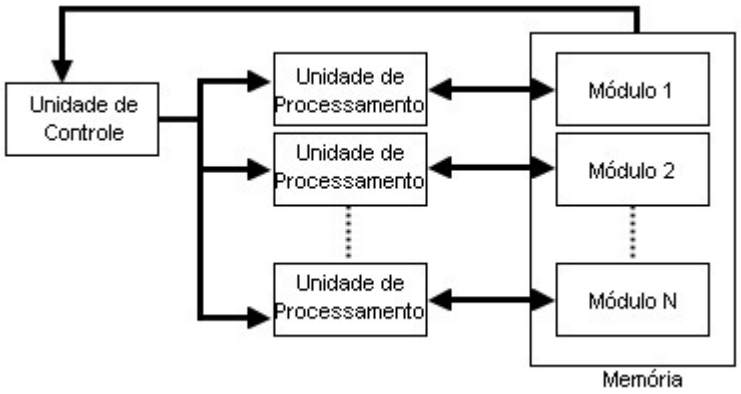

Figura 2 – Classe SIMD da Taxonomia de Flynn

Define o funcionamento de processadores vetoriais e matriciais. Diversos módulos de memória são necessários, as instruções seguem organizadas sequencialmente e possui uma unidade de controle e várias unidades de processamento. A Figura 2 ilustra essa classe.

MISD: Múltiplas instruções trabalhando no mesmo fluxo de dados. Esta classe da Taxonomia de Flynn é impossível de ser colocada em prática. A Figura 3 ilustra essa classe.

MIMD: Múltiplas instruções trabalhando em múltiplos dados. Possui várias unidades de controle, várias unidades de processamento e vários módulos de memória. Qualquer grupo de máquinas operando em conjunto, com interação entre elas, pode ser classificado como MIMD. A Figura 4 ilustra essa classe.

Além da Taxonomia de Flynn, as arquiteturas paralelas podem ser classificadas segundo o compartilhamento de memória. Multiprocessadores trabalham com vários processadores podendo acessar a mesma memória compartilhada, enquanto nos multicomputadores cada processador possui sua própria memória, fazendo necessário o uso de uma rede de interconexão para trocar informações.

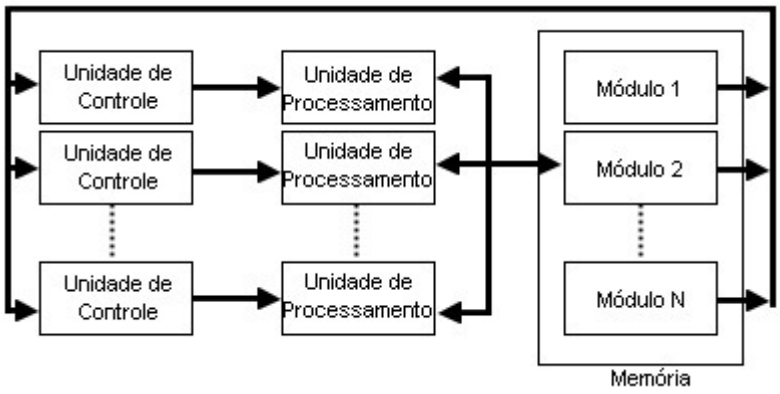

Figura 3 – Classe MISD da Taxonomia de Flynn

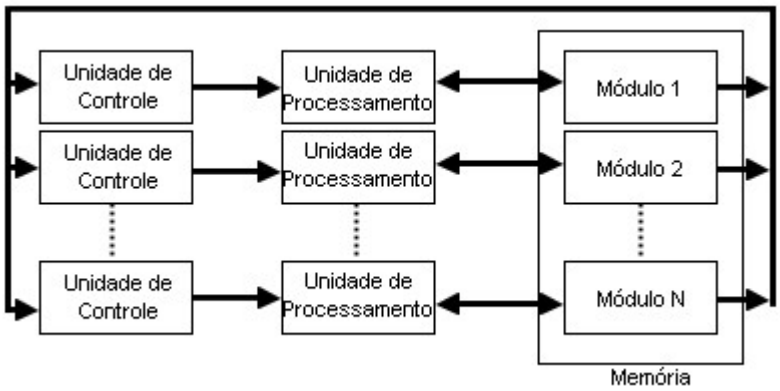

Figura 4 – Classe MIMD da Taxonomia de Flynn

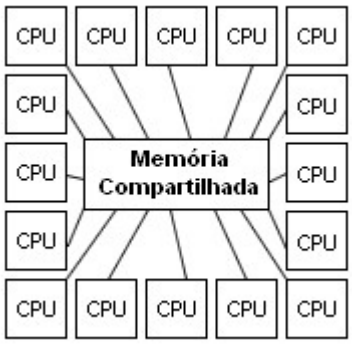

Figura 5 – Multiprocessador

2.1.1.1 Multiprocessadores

Multiprocessadores são sistemas nos quais múltiplas CPUs compartilham acesso à mesma memória. Uma propriedade que forma a base da comunicação entre processadores é: uma CPU escreve algum dado na memória e outra lê o mesmo dado. Sistemas multiprocessadores possuem algumas características únicas, como sincronização de processos e escalonamento, por exemplo. A Figura 5 exibe graficamente a arquitetura de um multiprocessador.

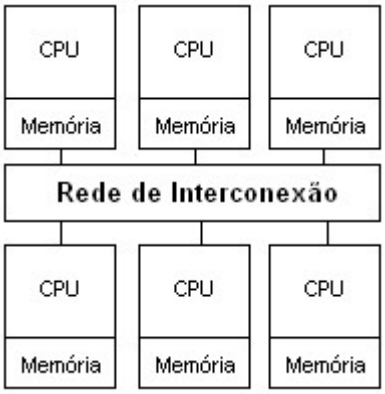

Figura 6 – Multicomputador

Alguns multiprocessadores possuem a característica de que uma certa palavra de memória possa ser lida na mesma velocidade que qualquer outra palavra. Essas máquinas são chamadas de *Uniform Memory Access* (UMA). Máquinas que não apresentam essa propriedade são chamadas de *Non-Uniform Memory Access* (NUMA) [Tanenbaum e Bos 2015].

2.1.1.2 Multicomputadores

Multicomputadores são sistemas nos quais cada CPU possui sua própria memória, não podendo ser diretamente acessada por nenhum outro processador. A troca de informações nesse sistema é feita através de uma rede de interconexão. A Figura 6 exibe graficamente a arquitetura de um multicomputador.

O acesso à memória nos multicomputadores é classificado como *Non-Remote Memory Access* (NORMA), afinal não é possível que uma CPU tenha acesso à memória remota [Hwang e Xu 1998].

2.1.1.3 Aceleradores

Uma *General Purpose Graphics Processing Unit* (GPGPU), ou um acelerador, é uma placa gráfica que pode ser usada para computação de propósito geral. São classificados como SIMD pela Taxonomia de Flynn, permitindo que vários dados possam ser processados em paralelo. As placas atuais utilizam uma extensão desse conceito, chamada de *Single Instruction, Multiple Threads* (SIMT), garantindo a execução da mesma instrução em threads diferentes. A Figura 7 exibe graficamente a arquitetura de um acelerador.

GPGPUs são compostas por vários processadores, que possuem vários núcleos cada, sendo que cada processador possui sua própria memória interna, compartilhada pelas suas *Arithmetic Logic Units* (ALUs), e uma memória compartilhada entre todos os processadores [Miranda 2010].

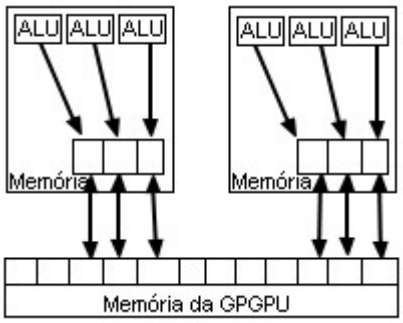

Figura 7 – Acelerador

2.1.2 Programação Paralela

A seguir serão apresentados os principais meios de desenvolvimento de aplicações paralelas para multiprocessadores, multicomputadores e aceleradores.

2.1.2.1 Programação para Multiprocessadores

A principal API de desenvolvimento de aplicações paralelas para multiprocessadores é o *Open Multi-Processing* (OpenMP). O OpenMP é baseado em diretivas de compilação, podendo ser usado em C/C++ ou em Fortran, combinando regiões sequenciais e paralelas no mesmo código fonte. As diretivas permitem a criação de regiões paralelas nas quais múltiplas threads são criadas e executadas em paralelo. O número de threads em uma região paralela pode ser determinado pelo usuário. Todavia, em programas paralelos utiliza-se normalmente uma thread para cada núcleo de processamento da arquitetura.

Para C/C++ as diretivas são da seguinte maneira:

```
#pragma omp [diretiva] [atributos]
```

A Figura 8 exemplifica a soma de dois vetores utilizando o OpenMP. Neste exemplo, a diretiva `parallel` foi usada para iniciar uma região paralela, enquanto a diretiva `for` serviu para paralelizar o laço, dividindo a execução das iterações do mesmo em várias *threads*. Foram declarados 3 vetores, e depois os vetores `a` e `b` foram somados e o resultado da soma foi armazenado no vetor `c`.

Variáveis globais são compartilhadas entre as threads, porém variáveis criadas dentro de um laço são privadas. O sucesso do OpenMP se deve ao fato de que ele é bem simples de ser usado e, por conta de uma alta aceitação, consegue ser executado em várias plataformas diferentes [Chapman, Jost e Pas 2008].

2.1.2.2 Programação para Multicomputadores

A principal API de desenvolvimento de aplicações paralelas para multicomputadores é o MPI. O MPI permite que dados sejam transmitidos entre processos em um ambiente

```
1   int main(int argc, char *argv[]) {
2       int a[1000], b[1000], c[1000];
3
4       initialize_vectors(&a, &b);
5
6       #pragma omp parallel for
7       for (int i = 0; i < 1000; i++)
8           c[i] = a[i] + b[i];
9
10      return 0;
11  }
```

Figura 8 – Exemplo da soma de vetores em OpenMP.

de memória distribuída. As principais características do MPI são portabilidade do código fonte, implementação eficiente, várias funcionalidades e suporte à arquiteturas paralelas heterogêneas.

Programas escritos em Fortran ou C/C++ são compilados normalmente, porém ligados com a biblioteca MPI. O MPI fornece funções do tipo `send` e `receive` síncronas e assíncronas para fazer a comunicação entre os processos. Além disso, ele oferece funções específicas e otimizadas para comunicação em grupo. Basicamente, as funções de envio e recebimento recebem como entrada os dados a serem transmitidos (assim como seu tipo e a quantidade de dados), o destinatário/remetente, entre outras.

A Figura 9 exemplifica a soma de dois vetores utilizando o MPI. Nas linhas 2 a 3 é feita a declaração das variáveis necessárias, no caso os vetores `a`, `b`, `c` e duas variáveis de controle que serão inicializadas pelo MPI (linhas 7 a 8) e conterão o *rank* do processo MPI (`rank`) e a quantidade total de processos MPI (`size`). O ambiente paralelo do MPI é inicializado na linha 4. A linha 11 é somente executada pelo processo MPI cujo *rank* é igual à 0 e servirá para inicializar os vetores. As linhas 13 e 14 calculam a quantidade de elementos que cada processo irá receber. As linhas 16 a 19 usam o `Scatter`, que é uma função do MPI que divide o vetor em partes iguais, no caso com a quantidade de elementos calculada anteriormente, e envia cada uma dessas partes para um processo diferente. As linhas 21 a 23 somam os vetores divididos e salvam o resultado num vetor temporário. Cada processo terá seu próprio resultado, independente dos outros processos. Nas linhas 25 e 26 é utilizada a função `Allgather`, que faz cada processo enviar sua parte do resultado calculado para todos os outros processos, juntando todas as partes e formando o vetor completo que contém o resultado final da soma, no caso o vetor `c`. A linha 28, por fim, encerra o ambiente paralelo do MPI.

O MPI não é uma implementação, mas sim uma especificação, possuindo diversas implementações diferentes. Uma das mais utilizadas implementações que existem atualmente é o OpenMPI, uma biblioteca de código aberto que implementa o MPI [Gropp, Lusk e Skjellum 1999].

```
1   int main(int argc, char *argv[]) {
2       int a[1000], b[1000], c[1000];
3       int rank, size;
4
5       MPI_Init(&argc, &argv);
6
7       MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8       MPI_Comm_size(MPI_COMM_WORLD, &size);
9
10      if(rank == 0)
11          initialize_vectors(&a, &b);
12
13      int elements_per_proc = 1000 / size;
14      int sub_a[elements_per_proc], sub_b[elements_per_proc];
15
16      MPI_Scatter(a, elements_per_proc, MPI_INT, &sub_a,
17                  elements_per_proc, MPI_INT, 0, MPI_COMM_WORLD);
18      MPI_Scatter(b, elements_per_proc, MPI_INT, &sub_b,
19                  elements_per_proc, MPI_INT, 0, MPI_COMM_WORLD);
20
21      int temp[elements_per_proc];
22      for (i = 0; i < elements_per_proc; i++)
23          temp[i] = sub_a[i] + sub_b[i];
24
25      MPI_Allgather(&temp, elements_per_proc, MPI_INT,
26                    c, elements_per_proc, MPI_INT, MPI_COMM_WORLD);
27
28      MPI_Finalize();
29
30      return 0;
31  }
```

Figura 9 – Exemplo da soma de vetores em MPI

2.1.2.3 Programação para Aceleradores

A principal API de desenvolvimento para aceleradores é o *Compute Unified Device Architecture* (CUDA), tecnologia da NVIDIA. A ideia é que os desenvolvedores possam usar o processamento da GPU para computação de propósito geral. A palavra chave `global` mostra para o compilador que a função a seguir será executada na GPU. O CUDA também oferece funções para alocar dinamicamente dados na memória da GPU, podendo alocar com a função `cudaMalloc()` e depois liberar com a função `cudaFree()` [Sanders e Kandrot 2010].

A Figura 10 exemplifica a soma de dois vetores, vistos anteriormente em OpenMP e MPI, porém utilizando a tecnologia CUDA. As linhas 1 a 5 utilizam o prefixo `global`, cujo objetivo é informar que o código da função será executado na GPU. A função `add` é chamada da seguinte forma: $add \lll N, 1 \ggg$, onde o parâmetro `N` é o número de blocos que serão criados pela GPU para executar em paralelo. A variável `blockIdx.x` informa qual bloco está sendo executado. O código `if (tid < 1000)` garante que não haverá acesso ao lixo de memória, no caso de existir mais blocos de GPU do que elementos para serem calculados. As linhas 8 e 9 declaram os vetores e variáveis necessários. As linhas 11 a 13 são responsáveis por alocar a memória da GPU utilizando a função `cudaMalloc()`. As linhas 17 a 20 copiam os vetores `a` e `b` para a GPU. A linha 22 chama a função `add`

```
1   __global__ void add(int *a, int *b, int *c) {
2       int tid = blockIdx.x;
3       if (tid < 1000)
4           c[tid] = a[tid] + b[tid];
5   }
6
7   int main( void ) {
8       int a[1000], b[1000], c[1000];
9       int *dev_a, *dev_b, *dev_c;
10
11      HANDLE_ERROR(cudaMalloc((void**)&dev_a, 1000 * sizeof(int)));
12      HANDLE_ERROR(cudaMalloc((void**)&dev_b, 1000 * sizeof(int)));
13      HANDLE_ERROR(cudaMalloc((void**)&dev_c, 1000 * sizeof(int)));
14
15      initialize_vectors(&a, &b);
16
17      HANDLE_ERROR(cudaMemcpy(dev_a, a, 1000 * sizeof(int),
18                              cudaMemcpyHostToDevice));
19      HANDLE_ERROR(cudaMemcpy(dev_b, b, 1000 * sizeof(int),
20                              cudaMemcpyHostToDevice));
21
22      add<<<1000,1>>>(dev_a, dev_b, dev_c);
23
24      HANDLE_ERROR(cudaMemcpy(c, dev_c, 1000 * sizeof(int),
25                              cudaMemcpyDeviceToHost));
26
27      cudaFree(dev_a);
28      cudaFree(dev_b);
29      cudaFree(dev_c);
30
31      return 0;
32  }
```

Figura 10 – Exemplo da soma de vetores em CUDA

que será executada na placa gráfica. As linhas 24 e 25 copiam o vetor `c` de volta para a CPU. As linhas 27 a 29 liberam a memória alocada na GPU usando a função `cudaFree()` e finalizam a execução do programa.

Para compilar o código usando o compilador CUDA C++ basta executar:

```
nvcc add.cu -o add_cuda
```

2.2 Criptografia de Dados

Do grego *Kriptos* (oculto) e *Grapho* (escrita), é o nome dado à ciência de codificar e decodificar mensagens. Tem como meta garantir:

- **Autenticação**: Identificar o remetente da mensagem;
- **Integridade**: Não adulteração da mensagem original;
- **Não Recusa**: Remetente não pode negar que enviou a mensagem.

A criptografia pode ser classificada como simétrica ou assimétrica, dependendo de como as chaves de codificação e decodificação são utilizadas. Há também o resumo

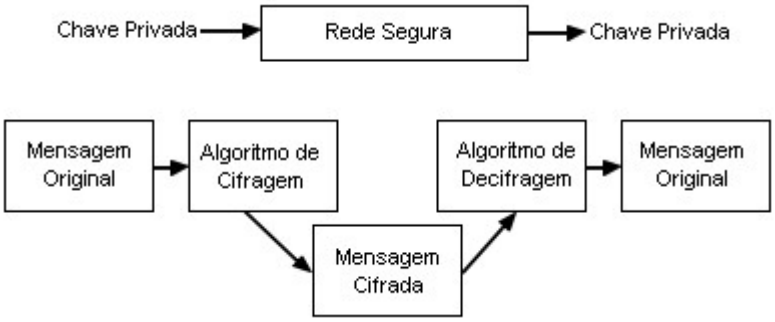

Figura 11 – Criptografia Simétrica

criptográfico, ou Hash, que é um número pequeno que representa todo um documento [Stallings 2014].

2.2.1 Criptografia Simétrica

O conceito mais antigo de criptografia é chamado de criptografia simétrica. Neste modelo a chave que dá acesso à mensagem é a mesma, tanto para codificar como para decodificar a mensagem, e deve permanecer em segredo, por isso é chamada de chave privada. A chave é utilizada para evitar que terceiros tenham acesso à mensagem, mesmo conhecendo o algoritmo utilizado e tendo em mãos a mensagem cifrada. A Figura 11 exibe o funcionamento da criptografia simétrica.

A maior vantagem da criptografia simétrica é sua facilidade de uso e velocidade para executar os algoritmos criptográficos. O problema deste modelo é que a chave usada para cifrar precisa ser compartilhada com o destinatário, abrindo uma brecha para terceiros interceptarem a chave [Stallings 2014].

Os principais algoritmos de criptografia simétrica são:

- *Advanced Encryption Standard* (AES): Desenvolvido pelo *National Institute of Standards and Technology*, é o algoritmo padrão usado pelo governo dos Estados Unidos da América. Possui um tamanho de bloco fixo em 128 bits, chave de 128, 192 ou 256 bits, rápido e fácil de executar e utiliza pouca memória.

- *Data Encryption Standard* (DES): Desenvolvido pela IBM em 1977, foi o algoritmo mais utilizado no mundo até a padronização do AES. Possui um tamanho de chave pequeno, de apenas 56 bits, o que possibilita quebrar o algoritmo por força bruta. A partir de 1993 passou a ser recomendada a utilização do 3DES, uma variação do DES no qual o ciframento é feito 3 vezes seguidas, porém é muito lento para se tornar um algoritmo padrão.

Figura 12 – Criptografia Assimétrica

2.2.2 Criptografia Assimétrica

Modelo desenvolvido pelo matemático Clifford Cocks, no qual as chaves para cifrar e decifrar são diferentes, chamadas de assimétricas. A chave pública pode ser vista por qualquer pessoa, porém a chave privada permanece em posse apenas do titular. Uma pessoa pode utilizar sua chave privada para decodificar uma mensagem criptografada com sua chave pública. A Figura 12 exibe o funcionamento da criptografia assimétrica.

A maior vantagem deste modelo é a segurança, uma vez que a chave privada não é compartilhada. Porém a velocidade é muito menor do que os algoritmos simétricos, o que pode não permitir o seu uso em algumas situações [Stallings 2014].

Os principais algoritmos de criptografia assimétrica são:

- *Rivest-Shamir-Adleman* (RSA): Desenvolvido em 1977 no *Massachusetts Institute of Technology* (MIT). É o algoritmo assimétrico mais utilizado no momento, além de ser um dos mais poderosos que existem. É baseado no fato de que dois números primos são facilmente multiplicados para gerar um terceiro número, porém é muito difícil recuperar esses números a partir do terceiro número. Para se descobrir a chave privada, é necessário fatorar números muito grandes, o que pode levar um tempo considerável. Assim, a segurança do RSA é baseada na dificuldade de fatoração de números primos grandes.

- *ElGamal Encryption System* (ELGAMAL): Baseado em grandes cálculos matemáticos. Sua segurança é baseada na dificuldade de calcular logaritmos discretos em um corpo finito.

- *Diffie-Hellman Key Exchange* (DH): Mais antigo dos métodos assimétricos, também é baseado no problema dos logaritmos discretos. Não é possível usá-lo para assinaturas digitais.

2.2.3 Resumo Criptográfico

Resumo criptográfico, também conhecido por hash, são funções criptográficas unidirecionais, ou seja, não é possível obter o conteúdo original a partir do hash. Uma característica dessas funções é que, independente do tamanho do texto, hash sempre terá um tamanho fixo, geralmente de 128 bits. Outra propriedade é que duas mensagens distintas não irão gerar o mesmo hash [Pfleeger, Pfleeger e Margulies 2015].

Os principais algoritmos de hash utilizados atualmente são:

- *Message Digest* (MD5): Desenvolvido por Ron Rivest, do MIT. Produz um hash de 128 bits. É um algoritmo rápido, simples e seguro, porém não é recomendado devido ao pequeno tamanho de 128 bits, sendo preferível um hash de maior valor.

- *Secure Hash Algorithm* (SHA): Criado pela *National Security Agency* (NSA), gera um hash de 160 bits. É recomendável o uso do SHA-2, uma variação mais forte e segura do que o SHA-1, devido ao maior número de bits que é gerado.

Alguns usos de funções de hash são:

- **Verificar integridade de arquivos**: Basta tirar o hash de um arquivo e guardá-lo. Em um momento futuro é possível tirar o hash novamente e comparar com o antigo, se forem iguais então o arquivo está íntegro.

- **Armazenamento de senhas**: A forma mais segura de armazenar uma senha é armazenar o hash da mesma, afinal não é possível obter a senha original. Quando precisar ser feita a verificação se uma senha digitada está correta, basta comparar o hash da senha digitada com o hash armazenado, se forem iguais então a senha está correta.

2.3 Compressão de Dados

Comprimir dados é o ato de reduzir o tamanho de arquivos, diminuindo o espaço que eles ocupam em disco e aumentando o desempenho de aplicativos que usam esses dados. Essa técnica é interessante para diversos fins, desde um usuário de smartphone que deseja armazenar mais fotos no seu aparelho até um serviço *web* que envia muitos dados através da internet. A Figura 13 mostra o conceito da compressão de dados.

Existem duas formas de compressão, com perdas e sem perdas, que descartam partes insignificantes do arquivo ou mantém todo o conteúdo, respectivamente.

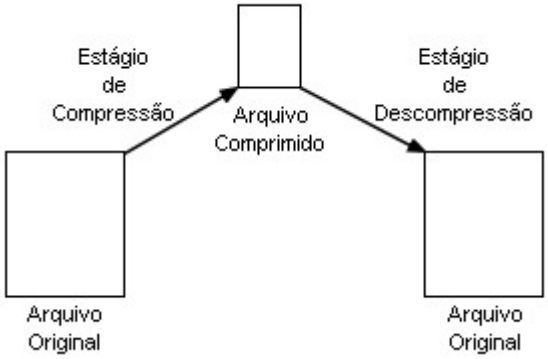

Figura 13 – Compressão de Dados

2.3.1 Compressão sem Perdas

A compressão sem perdas, ou *Lossless Data Compression*, garante que os dados obtidos após a descompressão serão exatamente iguais aos dados originais que foram comprimidos. Essa técnica é geralmente utilizada quando não se pode perder nada do conteúdo original, como arquivos de texto ou informações delicadas de experimentos científicos por exemplo.

2.3.2 Compressão com Perdas

A compressão com perdas, ou *Lossy Data Compression*, garante que os dados obtidos após a descompressão serão bastante parecidos com o conteúdo original, com diferenças mínimas. Essa técnica é geralmente utilizada para arquivos de áudio ou vídeo, nos quais a diferença de conteúdo é imperceptível e o tamanho é reduzido consideravelmente.

2.3.3 Compressão de Textos

A compressão de textos se baseia em representar o texto original de outra maneira, usando símbolos que ocupem menos espaço. Com isso se ganha também velocidade ao se fazer busca em grandes documentos. A desvantagem é o tempo necessário para a descompressão do conteúdo.

Um dos métodos mais conhecidos é o método de Huffman, de 1952. Nele, um código único é associado a cada caractere diferente do texto. Códigos menores são associados com os caracteres que aparecem com maior frequência. É o método mais eficiente para compressão de textos em linguagem natural, com 60% de redução no tamanho do arquivo. O método de Huffman elimina todos os espaços entre palavras. No momento da descompressão, a não ser que exista um separador, como uma vírgula, um espaço é inserido entre as palavras [Salomon 2007].

2.3.4 Compressão de Imagens

Existem formatos de imagens que comprimem com ou sem perdas. Os mais conhecidos formatos de compressão sem perdas são PNG, *Joint Photographic Experts Group 2000* (JPEG2000) e TIFF.

A compressão sem perdas explora a redundância entre pixeis e garante que nenhum dado será perdido. É especialmente importante em casos nos quais a fidelidade dos dados é muito importante, como para a fotografia profissional. Os algoritmos mais usados são o *Run Lenght Encoding* (RLE), *Lempel Ziv* (LZ), *Lempel Ziv Wech* (LZW) e o algoritmo de Huffman, o qual é usado nos formatos PNG e TIFF.

Dentre os métodos com perdas, os mais conhecidos são JPEG e GIF. A compressão com perdas busca eliminar detalhes que não são perceptíveis ao olho humano. Porém há formatos, como o GIF, que utilizam um grau maior de perda, causando uma degradação grande na imagem.

3 O Algoritmo de Cripto-Compressão GMPR

O algoritmo de cripto-compressão GMPR foi desenvolvido pela *Sheffield Hallam University* em MATLAB [Rodrigues e Siddeq 2016]. Neste projeto de trabalho de conclusão de curso, este algoritmo será reescrito em C++ para que seja possível paralelizá-lo.

O algoritmo GMPR foi desenvolvido com o foco em imagens, sendo superior em qualidade em relação ao conhecido formato JPEG e com qualidade equivalente ao formato JPEG2000. A Tabela 2 mostra uma comparação do algoritmo GMPR com os formatos JPEG e *Bitmap* (BMP) para imagens 3D.

Ele utiliza a *Discrete Cosine Transform* (DCT) e um algoritmo de minimização de matrizes no estágio de compressão, além de um novo método concorrente de busca binária no estágio de descompressão. Os principais passos do algoritmo de compressão consistem em dividir a imagem em blocos e aplicar a DCT para cada bloco, aplicar o algoritmo de minimização de matrizes nos coeficientes `AC` de cada bloco, reduzindo a matriz para 1/3 do tamanho, construir uma *Look Up Table* (LUT) para permitir a recuperação dos dados originais no estágio de descompressão, aplicar um delta na lista de coeficientes `DC` e aplicar a codificação aritmética nos resultados. O estágio de descompressão utiliza a LUT e o algoritmo de busca binária para recuperar todos os coeficientes `AC`, enquanto os coeficientes `DC` são recuperados revertendo a codificação aritmética, e, finalmente, a DCT inversa recupera a imagem original [Rodrigues e Siddeq 2016]. A Figura 14 ilustra o processo de compressão.

3.1 Compressão

Primeiramente a imagem é dividida em blocos de tamanho `n`, e então é aplicada a DCT em cada bloco. Cada bloco consiste de coeficientes `DC`, que são o valor médio do bloco, e os demais coeficientes, chamados `AC`. A DCT, um processo sem perdas e reversível, serve para identificar redundâncias na imagem, ou seja, pixeis muito semelhantes em relação aos seus vizinhos.

Em seguida é aplicado o algoritmo de minimização de matrizes na lista de coeficientes `AC`, eliminando todos os zeros, que geralmente são muitos, reduzindo seu tamanho e produzindo o vetor minimizado. São definidas 3 chaves que irão multiplicar cada 3 entradas deste vetor minimizado, e então os 3 valores são somados, ou seja, cada 3 valores do vetor é transformado em apenas 1, reduzindo o tamanho para 1/3 do original, produzindo o vetor minimizado codificado.

A etapa de descompressão inicia na recuperação dos coeficientes `DC`, revertendo

Imagem	BMP	JPEG	GMPR
Maçã	336 MB	52.4 MB	0.929 MB
Estátua	366 MB	58.5 MB	0.916 MB
Rosto	200.7 MB	45.5 MB	0.784 MB

Tabela 2 – Compressão de Imagens 3D.

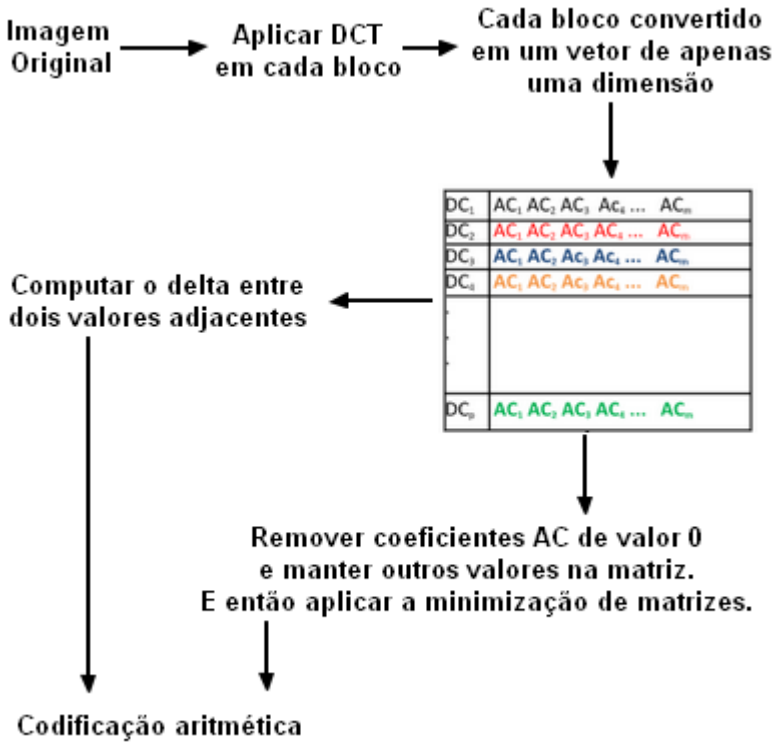

Figura 14 – Estágio de Compressão do GMPR

a codificação aritmética. Os coeficientes `AC` são recuperados através da busca binária, utilizando as 3 chaves geradas na compressão é possível recuperar o vetor minimizado, decodificando-o.

Em seguida os coeficientes `DC` a `AC` são combinados, recolocando todos os zeros que foram removidos na compressão, e em seguida aplicando a DCT inversa para recuperar a imagem original [Rodrigues e Siddeq 2016].

3.2 Criptografia

De modo geral, o algoritmo GMPR inicia a etapa de cifragem obtendo o texto plano e convertendo-o para `ASCII`. Em seguida é obtida uma lista dos caracteres contidos no arquivo, sem repetir nenhum. Por exemplo: o texto "abac" irá gerar uma lista contendo os caracteres "abc", não incluindo o caractere "a" duas vezes.

São geradas 3 chaves aleatórias. A partir disso é gerada uma lista chamada `nCoded`,

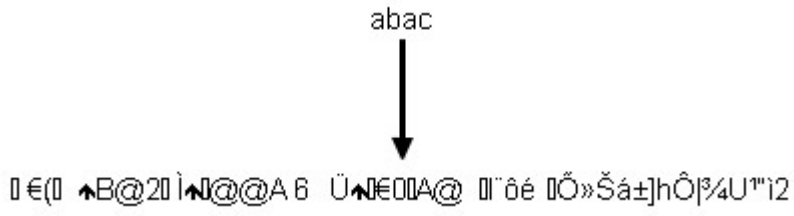

Figura 15 – Binário gerado pelo algoritmo GMPR

na qual a cada 3 caracteres do texto é gerado 1 elemento codificado usando as chaves geradas. O texto “abac” geraria 2 elementos nessa lista, pois “aba” seria codificado no primeiro elemento e “c” codificado no segundo elemento;

É criado um vetor chamado `codedvector` que é formado pelas 3 chaves, pela lista única de caracteres e pela lista `nCoded`. Em seguida o texto final codificado é gerado.

Para decifrar o conteúdo e obter o arquivo original, o algoritmo começa obtendo o texto cifrado e transformando-o em um vetor. Em seguida são extraídas as 3 chaves usadas no processo de cifragem, a lista única de caracteres e a lista `nCoded`. Por fim, o conteúdo é decodificado.

Se o arquivo cifrado com o algoritmo GMPR for aberto em algum editor de texto será mostrado algo parecido com o exemplo da Figura 15. As seções a seguir apresentam os pseudo-códigos das etapas de cifragem e decifragem do algoritmo.

3.2.1 Algoritmo de Cifragem

Nas linhas 2 a 4 é criada e preenchida a `nLimited`, uma lista contendo todos os caracteres únicos do texto plano.

Em seguida as 3 chaves aleatórias são geradas nas linhas 5 a 7. `K[0]` é gerada entre 0 e 1000. `K[1]` gera um valor entre 0 e 10 e depois multiplica pelo dobro do maior caractere do texto plano. `K[2]` gera um valor entre 0 e 10, multiplica pela chave `K[1]` e depois multiplica pelo maior caractere do texto plano.

Nas linhas 8 a 10 é criada e preenchida a `nCoded`, uma lista que contém as chaves `K` multiplicadas pelo texto plano. Nas linhas 11 a 15 é criado e preenchido o `codedvector`, um vetor que contém as chaves `K`, o tamanho da lista `nLimited`, o tamanho da lista `nCoded`, o conteúdo da lista `nLimited` e o conteúdo da lista `nCoded`.

Na linha 16 é convertido o vetor `codedvector` para `string`.

Na linha 17 é obtida uma lista de intervalos de probabilidades para cada símbolo do texto plano. Em seguida são aplicados os intervalos à cada elemento do texto e adicionados ao texto final codificado nas linhas 18 e 19. Por fim o arquivo é salvo em disco na linha 20.

Algorithm 1 Cifragem

```
 1: nData ← abrirArquivoOriginal()
 2: for i from 0 to nData.tamanho() do
 3:     if !(nLimited ⊃ nData[i]) then
 4:         adicionarElemento(nLimited, nData[i])
 5: K[0] ← rand() % 1001
 6: K[1] ← (rand() % 11) * 2 * maxvalue
 7: K[2] ← K[1] * maxvalue * (rand() % 11)
 8: while i <= nData.tamanho() do
 9:     nCoded[i] ← K[0] * nData[i] + K[1] * nData[i + 1] + K[2] * nData[i + 2]
10:     i ← i + 3
11: adicionarElemento(codedvector, K)
12: adicionarElemento(codedvector, nLimited.tamanho())
13: adicionarElemento(codedvector, nCoded.tamanho())
14: adicionarElemento(codedvector, nLimited)
15: adicionarElemento(codedvector, nCoded)
16: string msg ← paraString(codedvector)
17: stats ← obterListaIntervaloProbabilidades()
18: for i from 0 to nData.tamanho() do
19:     AplicaProbabilidades(nData[i], stats)
20: EscreverDisco(stats)
```

3.2.2 Algoritmo de Decifragem

Na linha 1 a rotina `ArDecodeFile` abre o arquivo cifrado, faz uma leitura do cabeçalho e gera a lista de probabilidades que será usada para decodificar o arquivo. Em seguida a `string` é convertida para vetor na linha 2.

Nas linhas 3 a 5 são obtidas as 3 chaves que foram usadas para cifrar. O tamanho da lista de caracteres únicos `nLimited` é obtido na linha 6 e seu conteúdo obtido nas linhas 8 e 9. O tamanho da lista `nCoded`, que contém as chaves `K` multiplicadas pelo texto plano, é obtido na linha 7 e seu conteúdo obtido nas linhas 10 e 11.

Nas linhas 12 a 20 o texto é decodificado, obtendo o conteúdo original. É feito o processo inverso da geração da lista `nCoded` no processo de cifragem, e quando o elemento codificado é encontrado então ele é salvo na lista `nDecoded`. Por fim o arquivo é salvo em disco.

Algorithm 2 Decifragem

```
1: msg ← ArDecodeFile()
2: data ← paraVetor(msg)
3: K[0] ← data[0]
4: K[1] ← data[1]
5: K[2] ← data[2]
6: nL ← data[3]
7: nC ← data[4]
8: for i from 5 to 5 + nL do
9:     nLimited[i] ← data[i]
10: for i from 5 + nL + 1 to 5 + nL + 1 + nC do
11:     nCoded[i] ← data[i]
12: ultimaPos ← data.tamanho()
13: for r from 0 to ultimaPos do
14:     for i from 0 to nL do
15:         for j from 0 to nL do
16:             for k from 0 to nL do
17:                 if data[r] == K[0] * nLimited[i] + K[1] * nLimited[j] + K[2] * nLimited[k] then
18:                     nDecoded[3 * r + 0] ← nLimited[i]
19:                     nDecoded[3 * r + 1] ← nLimited[j]
20:                     nDecoded[3 * r + 2] ← nLimited[k]
21: EscreverDisco(nDecoded)
```

4 Proposta

O algoritmo descrito no capítulo anterior possui várias possibilidades de ganho de desempenho. Todavia, para que isso seja possível, uma versão em C/C++ precisará ser desenvolvida.

Este trabalho de conclusão de curso possui três etapas principais. Na primeira etapa, será feita uma implementação em C++ do algoritmo de cripto-compressão GMPR. Com base nesta solução proposta, na segunda etapa serão estudadas formas de paralelizar o código para obter-se um desempenho ainda melhor. As APIs OpenMP, MPI e/ou CUDA serão exploradas para implementar soluções paralelas e/ou distribuídas para este problema. Por fim, na terceira etapa, serão realizados experimentos para medir o desempenho das soluções paralelas propostas.

No presente momento, somente as etapas de cifragem/decifragem foram implementadas em C++. As etapas de compressão/descompressão serão implementadas ao longo do primeiro semestre de 2018. A versão atual do código em C++ está disponível no GitHub[1].

A seguir, serão apresentadas algumas ideias iniciais de paralelização das etapas de cifragem/decifragem.

4.1 Paralelismo com Multiprocessadores

A primeira ideia de paralelização do código é a utilização do OpenMP para executar as partes mais pesadas da aplicação em múltiplos núcleos do processador. Conforme visto no Capítulo 3, o algoritmo possui muitos laços do tipo `for`, o que permite a utilização da biblioteca OpenMP para executar esses laços em paralelo (através da diretiva `omp parallel for`), garantindo a correta execução das operações, porém em menos tempo.

Para isso, o objetivo inicial será definir em quais partes do código o programa leva mais tempo para executar, analisar se é possível quebrar essa parte em pedaços pequenos e atribuir múltiplas `threads` para a execução. Para isso, alguns experimentos preliminares foram feitos utilizando-se a ferramenta `Valgrind`[2] no Linux. A Figura 16 apresenta os resultados obtidos para o algoritmo de codificação, enquanto a Figura 17 apresenta os resultados para a decodificação.

Como é possível observar nas Figuras 16 e 17, os métodos mais custosos em termos de tempo de execução são o `generate_limited_data()` para a parte de codificação, o qual utiliza o método `is_in()` para verificar se algum caractere está presente na lista

[1] <https://github.com/leandroperin/ParallelCryptoCompression>
[2] <http://valgrind.org/>

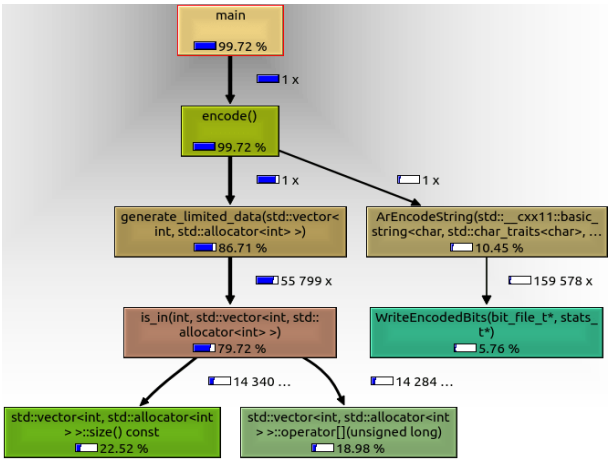

Figura 16 – Resultados do Valgrind para a Codificação.

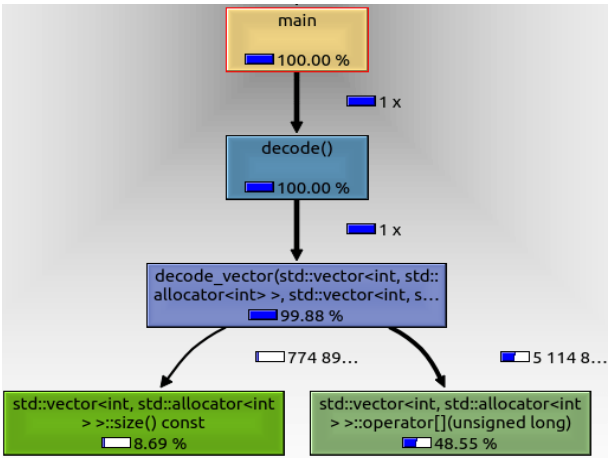

Figura 17 – Resultados do Valgrind para a Decodificação.

única de caracteres, e o `decode_vector()` para a parte de decodificação, o qual usa o operador `[]` para acessar os elementos do vetor que contém os dados codificados.

O próximo passo é então fazer uso de regiões paralelas para dividir a computação executada nos métodos anteriormente citados a fim de melhorar o desempenho geral da solução. Após a definição das regiões paralelas, será necessário analisar se existem variáveis que serão compartilhadas entre as *threads* e se há dependência entre elas. As cláusulas `shared` e `private` permitem descrever quais variáveis são compartilhadas entre todas as *threads* e quais são privadas em cada *thread*, respectivamente. Também, será verificada a existência de condições de corrida no código paralelizado. Nesses casos, regiões críticas do código, que são regiões que só podem ser executadas em uma *thread* de cada vez, deverão ser protegidas com o uso da diretiva `omp critical`.

Por fim, será avaliada a possibilidade de uma solução híbrida com uso de OpenMP e MPI. Esta solução permitiria a execução da versão paralela da aplicação em `clusters`.

4.2 Paralelismo com Aceleradores

Tendo em vista os resultados dos experimentos iniciais mostrados nas Figuras 16 e 17, uma possibilidade seria a implementação de *kernels* CUDA ou *Open Computing Language* (OPENCL), uma solução de código aberto, para realizar a computação dos métodos mais custosos em GPU. Neste caso, como mostrado no exemplo da Figura 10, será necessário implementar os *kernels*, alocar memória na GPU, copiar os dados que estão na CPU e, por fim, liberar a memória utilizada.

As GPUs trabalham muito bem com operações matriciais, portanto uma possibilidade de melhora do código seria armazenar o conteúdo em matrizes e executar o algoritmo de cripto-compressão diretamente nas placas gráficas, garantindo um ganho de desempenho. GPUs possuem centenas de núcleos, e essa arquitetura massivamente paralela pode fazer o algoritmo GMPR executar em um tempo consideravelmente menor.

Conteúdos que ganhariam bastante com o uso de GPUs são imagens e vídeos, afinal já são naturalmente armazenados em forma de matriz. A utilização do CUDA ou do OPENCL seria bastante eficaz para esse tipo de conteúdo.

4.3 Testes de Desempenho

Após a implementação das soluções propostas (OpenMP, MPI e/ou CUDA/OPENCL) serão feitos inúmeros testes e medições de desempenho, para que seja descoberta a melhor e mais rápida maneira de executar o algoritmo, garantindo o menor tempo de execução possível.

5 Cronograma

As atividades previstas no projeto estão descritas abaixo:

- **A1: Estudo da fundamentação teórica.** Nesta etapa será feito o estudo de toda a fundamentação teórica do trabalho, seus conceitos e formas de colocá-los em prática.
- **A2: Produção do código sequencial.** Nesta etapa será refatorado todo o código do algoritmo, produzindo uma versão limpa e sequencial do mesmo.
- **A3: Elaboração da proposta.** Nesta etapa será elaborada a proposta do trabalho, incluindo os meios de paralelizar e melhorar o desempenho do algoritmo.
- **A4: Escrita do relatório do TCC I.** Nesta etapa será escrito o relatório do TCC I, que inclui a fundamentação teórica, descrição do algoritmo sequencial e proposta de paralelização. Entrega prevista para o mês de fevereiro.
- **A5: Implementação da proposta.** Nesta etapa serão implementadas as ideias propostas para a paralelização do algoritmo.
- **A6: Realização de experimentos.** Nesta etapa serão realizados experimentos com a versão paralela do código, fazendo medições de desempenho no mesmo.
- **A7: Escrita do rascunho do TCC II.** Nesta etapa será escrito o rascunho do TCC II, que inclui a proposta implementada e os resultados obtidos. Previsão de entrega para o mês de maio.
- **A8: Preparação da defesa pública.** Nesta etapa será elaborada a defesa pública do trabalho.
- **A9: Defesa pública.** Nesta etapa será realizada a defesa pública do trabalho desenvolvido. Data prevista para o mês de junho.
- **A10: Entrega da versão final do TCC II.** Nesta etapa serão feitas correções e ajustes solicitados no trabalho. Entrega da versão final prevista para o mês de julho.

A Figura 18 apresenta o cronograma previsto para a realização das atividades descritas anteriormente, o que inclui atividades durante o segundo semestre de 2017 e o primeiro semestre de 2018.

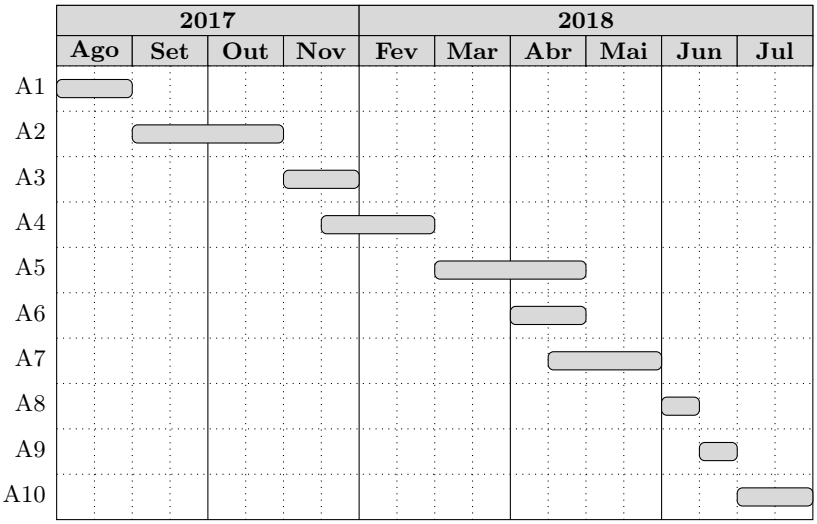

Figura 18 – Cronograma de Atividades

6 Conclusão

O algoritmo GMPR funciona corretamente, cumprindo seu objetivo, porém apresenta um desempenho insatisfatório. Este trabalho busca analisar o código e melhorá-lo, garantindo que seu tempo de execução diminua e o torne possível de ser utilizado em alguma aplicação real.

A utilização das tecnologias OpenMP, MPI e/ou CUDA/OPENCL aparentam ser as melhores opções disponíveis para melhorar o tempo de execução do algoritmo de cripto-compressão GMPR. Elas serão implementadas e terão os resultados medidos, como forma de garantir os ganhos de desempenho.

Ter o algoritmo funcionando de forma eficiente é uma conquista muito interessante, pois o mesmo pode ser aplicado em serviços *web* e na nuvem, reduzindo o armazenamento utilizado, os riscos de segurança e também o tempo necessário para enviar informações através da Internet.

O uso deste trabalho em projetos futuros é bastante possível, afinal esse algoritmo pode ser usado em qualquer projeto que tenha restrições de segurança das informações ou restrições de armazenamento. Há também a possibilidade desse projeto ser estendido para outras áreas, como a cripto-compressão de um banco de dados, dentre outras.

Referências

CHAPMAN, B.; JOST, G.; PAS, R. *Using OpenMP: Portable Shared Memory Parallel Programming*. [S.l.]: MIT Press, 2008. 353 p. ISBN 978-0262533027. Citado na página 19.

FLYNN, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, IEEE, v. 21, n. 9, p. 948–960, September 1972. Disponível em: <http://ieeexplore.ieee.org/document/5009071/>. Citado na página 15.

GROPP, W.; LUSK, E.; SKJELLUM, A. *Using MPI: Portable Parallel Programming with the Message-passing Interface*. [S.l.]: MIT Press, 1999. 371 p. ISBN 978-0262571326. Citado na página 20.

HWANG, K.; XU, Z. *Scalable Parallel Computing: Technology, Architecture, Programming*. First. [S.l.]: McGraw-Hill Science/Engineering/Math, 1998. 832 p. ISBN 978-0070317987. Citado na página 18.

MIRANDA, C. S. A evolução da gpgpu: arquitetura e programação. p. 1–7, 2010. Disponível em: <http://www.ic.unicamp.br/~ducatte/mo401/1s2010/T2/070498-t2.pdf>. Citado na página 18.

PFLEEGER, C. P.; PFLEEGER, S. L.; MARGULIES, J. *Security in Computing*. [S.l.]: Prentice Hall, 2015. 910 p. ISBN 978-0134085043. Citado na página 25.

RODRIGUES, M. A.; SIDDEQ, M. M. Information systems: Secure access and storage in the age of cloud computing. *Athens Journal of Sciences*, Athens Institute for Education and Research, v. 3, n. 4, p. 267–284, September 2016. Disponível em: <http://shura.shu.ac.uk/13715/>. Citado 4 vezes nas páginas 11, 12, 29 e 30.

SALOMON, D. *Data Compression: The Complete Reference*. [S.l.]: Springer Science Business Media, 2007. 1092 p. ISBN 978-1846286032. Citado 2 vezes nas páginas 11 e 26.

SANDERS, J.; KANDROT, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. [S.l.]: Addison-Wesley Professional, 2010. 312 p. ISBN 978-0132180139. Citado na página 21.

STALLINGS, W. *Cryptography and Network Security: Principles and Practice*. [S.l.]: Pearson, 2014. 731 p. ISBN 978-0133354690. Citado 3 vezes nas páginas 11, 23 e 24.

TANENBAUM, A. S.; BOS, H. *Modern Operating Systems*. Fourth. [S.l.]: Pearson, 2015. 1137 p. ISBN 978-0-13-359162-0. Citado 2 vezes nas páginas 15 e 18.