Parallel Implementations of the CSBP Stereo Vision Algorithm

Lucas Veronese, Lauro J. Lyrio Junior, Jorcy de Oliveira Neto, Avelino Forechi, Claudine Badue, Alberto F. De Souza

Laboratório de Computação de Alto Desempenho (LCAD)
Departamento de Informática (DI), Universidade Federal do Espírito Santo (UFES)
Av. Fernando Ferrari, 514 – 29060-970 – Vitória-ES – Brazil
{lucas.veronese, laurojljunior, jorcyd, avelino, claudine, alberto}@lcad.inf.ufes.br

Abstract

We developed two parallel versions of the Constant Space Belief Propagation algorithm (CSBP - one of the best stereo algorithms currently known) [1]: one in OpenMP and one in C+CUDA. For images with 640x480 pixels, the sequential version has a performance of 1.16 frames per second (FPS), the OpenMP parallel version has a performance of 3.7 FPS, while the C+CUDA version has a performance of 17.3 FPS in high-performance desktop machines. These results are important because they enable the implementation of autonomous vehicles with sensors like camera, which is one of the objectives of a PRONEX project being currently developed in LCAD-DI/UFES. One of the goals of this project is to implement an autonomous vehicle from a commercial automobile.

1. Introdução

As imagens projetadas dentro de nossos olhos mudam todo o tempo por conta do movimento dos olhos ou do nosso corpo como um todo. Contudo, em um aparente paradoxo, percebemos o mundo retratado nas imagens capturadas pelos olhos como estável. Além disso, as imagens projetadas nas retinas humanas são bidimensionais; entretanto, o cérebro é capaz de sintetizar uma representação tridimensional estável a partir delas (o que vemos, Figura 1), com informações sobre cor, forma e profundidade a respeito dos objetos no ambiente ao nosso redor, eliminando os efeitos dos movimentos dos olhos e do corpo.

O sistema visual biológico viabiliza a nossa movimentação através do ambiente 3D de forma precisa. Assim, a compreensão e a modelagem de funcionalidades relevantes do sistema visual biológico, como aquelas que nos permitem ver em três dimensões, podem contribuir para o desenvolvimento de sistemas de localização e mapeamento simultâneos (*Simultaneous Localization And Mapping* – SLAM [2]) e de navegação de veículos autônomos.

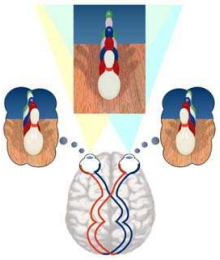

Figura 1: Criação, pelo cérebro, de representação 3D a partir de imagens 2D.

SLAM é talvez o problema mais fundamental da robótica autônoma. Veículos robóticos autônomos necessitam saber onde estão em sua área de atuação e como esta (sua área de atuação) está configurada para que possam navegar nela e realizar suas atividades de interesse. Para isso, estes veículos precisam usar sensores que captem informações do ambiente em quantidade e qualidade suficiente para realizar o mapeamento relevante e se localizar nos mapas gerados. Atualmente, um dos sensores mais utilizados são lasers de varredura (*Laser Range Scan*, ou *Light Detection And Ranging* - LIDAR).

Sensores LIDAR empregam feixes de laser para medir a distância de pontos ao longo de uma linha à frente usando um mecanismo mecânico/ótico de varredura – tal mecanismo não encontra paralelo na biologia. Além disso, sensores LIDAR são fortemente afetados por chuva, entre outras condições meteorológicas naturais, o que limita sua aplicabilidade em casos que requeiram operação ao ar livre, e são facilmente detectáveis à distância (por causa do laser), o que limita sua aplicabilidade militar.

Câmeras digitais, por outro lado, são hoje capazes de capturar imagens com milhões de pontos, possuindo preço substancialmente menor que LIDARs. Contudo, para empregar câmeras digitais na solução do problema de SLAM em tempo real, é necessário processar a enorme quantidade de dados disponibilizada pelas mesmas na forma de imagens de maneira equivalente ao nosso cérebro; i.e., é necessário sintetizar representações tridimensionais estáveis a partir de imagens bidimensionais.

Para transformar as imagens bidimensionais em nuvens de pontos 3D, são necessários algoritmos que encontrem a projeção dos mesmos pontos de um objeto no mundo 3D em imagens capturadas por duas ou mais câmeras. Com a informação sobre a localização destes pontos nas diversas imagens, e com o conhecimento sobre a geometria de cada câmera empregada e do posicionamento delas, é possível resolver o problema de percepção do mundo em 3D para pontos que sejam visualizados por duas ou mais câmeras. Técnicas que resolvem o problema de visão artificial 3D desta forma são conhecidas como técnicas de visão estéreo [1, 3].

Para viabilizar a navegação robótica utilizando câmeras digitais, os algoritmos de visão estéreo devem executar em tempo real, isto é, em no máximo 15 FPS. Neste trabalho de pesquisa, desenvolvemos duas versões paralelas do algoritmo *Constant Space Belief Propagation* (CSBP - um dos melhores algoritmos de visão estéreo atualmente conhecidos [4]) [1]: uma em OpenMP e outra em C+CUDA. Para imagens de 640x480 *pixels*, a versão sequencial tem desempenho de 1,16 FPS, a versão paralela em OpenMP tem desempenho de 3,7 FPS, enquanto que a versão paralela em C+CUDA tem desempenho de 17,3 FPS em máquinas desktop de alto desempenho. Estes resultados são importantes porque viabilizam a implementação de veículos autônomos com sensores do tipo câmera, o que é um dos objetivos do projeto PRONEX, sendo atualmente desenvolvido no Laboratório de Computação de Alto Desempenho (LCAD) do Departamento de Informática (DI) da Universidade Federal do Espírito Santo (UFES) – LCAD-DI/UFES. Uma das metas deste projeto é implementar um veículo autônomo a partir de um automóvel comercial.

Este artigo está organizado da seguinte forma. Após esta introdução, na Seção 2 apresentamos o algoritmo CSBP. Nas Seções 3 e 4, apresentamos nossas implementações paralelas em OpenMP e C+CUDA do algoritmo CSBP. Na Seção 5, descrevemos a metodologia experimental e, na Seção 6, discutimos os resultados experimentais. Finalmente, na Seção 7, apresentamos nossas conclusões.

2. CSBP Stereo

O algoritmo CSBP busca resolver o problema de visão estéreo por meio de *Stereo Matching* via *Belief Propagation* (BP) [5, 6]. Este é um algoritmo baseado em troca de mensagens muito utilizado para inferência em modelos gráficos como Redes Bayesianas e Campos Aleatórios de Markov (*Markov Random Field* – MRF).

Uma variação dos algoritmos baseados em BP é o *Hierarchical Belief Propagation* (HBP) [1]. Este algoritmo difere-se do BP tradicional por trabalhar de forma hierárquica, de um nível de grão grosso para um nível de grão fino, ou seja, a cada iteração do algoritmo, a resolução espacial do modelo gráfico é reduzida até atingir um nível mínimo, onde o algoritmo começa a convergir.

O algoritmo HBP [1] utiliza-se da variante *max-product* BP [7]. O algoritmo *max-product* BP efetua troca de mensagens em um modelo de grafo que define um *grid* 4-conectado em uma imagem. Cada mensagem é um vetor de dimensão dada pelo número de possíveis rótulos de disparidades **L**, e a cada iteração novas mensagens são computadas segundo a Equação 1:

$$M^t_{\mathbf{X},\mathbf{Y}}(d) = \underset{d_{\mathbf{X}}}{\operatorname{argmin}}(E_{D,\mathbf{X}}(d_{\mathbf{X}}) + \sum_{s \in N(\mathbf{X}), \mathbf{X} \neq \mathbf{Y}} M^{t-1}_{s,\mathbf{X}}(d_{\mathbf{X}}) + h(d_{\mathbf{X}}, d)), \quad (1)$$

onde $M^t_{\mathbf{X},\mathbf{Y}}$ é o vetor de mensagens passado do pixel **X** para um de seus vizinhos **Y** na iteração **t**, $E_{D,\mathbf{X}}$ é o termo de dados do pixel **X**, $h(d_{\mathbf{X}}, d)$ é o custo de deslocamento do pixel, e *d* é o rotulo de disparidade que minimiza a energia total do pixel **X**, que já contém o termo de dados $E_{D,\mathbf{X}}$ e cujo termo de suavização é

$$\begin{aligned} E_{\mathbf{X}}(d) &= E_{D,\mathbf{X}}(d) + E_{S,\mathbf{X}}(d) \\ &= E_{D,\mathbf{X}}(d) + \sum_{\mathbf{Y} \in N(\mathbf{X})} M_{\mathbf{Y},\mathbf{X}}(d). \end{aligned} \quad (2)$$

A etapa de refinamento das mensagens de um pixel **X** de um nível de grão mais grosso para um nível de grão mais fino é dada por:

$$M_{\mathbf{X}'_i,\mathbf{Y}'_{i,j}} = M_{\mathbf{X},\mathbf{Y}_j}, i, j \in [1, 4], \quad (3)$$

onde $\mathbf{Y'_{i,j}}$ são os quatro vizinhos do pixel $\mathbf{X'_i}$ e Y_j são os 4 vizinhos correspondentes ao pixel **X**, como pode ser observado na Figura 2.

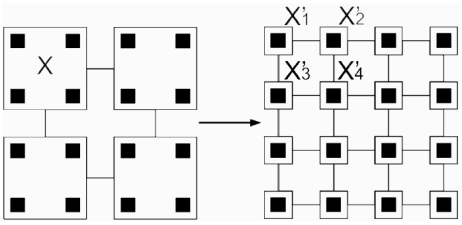

Figura 2: Ilustração de dois níveis de refinamento, onde cada nó X na figura à esquerda corresponde a um bloco de 4 pixels na figura à direita.

Na abordagem do trabalho apresentado em [1], foi utilizado o algoritmo de HBP descrito anteriormente, onde as mensagens são atualizadas em cada nível e então propagadas para o nível de grão mais fino já refinadas. Porém, no trabalho apresentado em [8], foi verificado que, na média, somente uma pequena parte dos níveis de disparidade e suas mensagens correspondentes são necessários em cada pixel da imagem para reconstruir as mensagens de BP sem muita perda. Como resultado, o trabalho apresentado em [1] não somente aplica uma abordagem de refinamento no domínio espacial, mas também no domínio de profundidade, reduzindo gradualmente o número de níveis de disparidades quando é feita a propagação das mensagens de um nível de grão mais grosso para um nível de grão mais fino. Tais modificações deram origem a um novo algoritmo de complexidade de espaço constante *O(1)*, denominado *Constant Space Belief Propagation* (CSBP) [1], que será o alvo de estudo em nosso trabalho.

Algoritmo 1: Constant-Space Belief Propagation	
1	Calcule o termo de dados E_D no nível mais baixo (nível S - 1)
2	Apenas os k_{S-1} custos de dados são selecionados e passados ao passo 5 em cada pixel. Os outros custos de dados e os níveis de disparidades correspondentes são tratados isoladamente.
3	Inicialize as mensagens com zero
4	**Para** $s = S - 1$ ate 0 **faça**
5	Atualize iterativamente o vetor de mensagens em cada pixel e para cada disparidade de acordo com a Equação 1.
6	**Se** s > 0 **então**
7	Inicialize as mensagens para o próximo nível usando a Equação 3 e selecione os níveis de disparidade.
8	Calcule o termo de dados para o próximo nível, usando os níveis de disparidade selecionados.
9	Calcule a energia total de cada pixel para cada disparidade candidata de acordo com a Equação 2.
10	Selecione $k_{s-1} = k_s/2$ níveis de disparidade e valores de mensagem correspondente aos k_{s-1} menores valores de energia e aplique a estes o passo numero **5** para cada pixel. Os outros valores de disparidades e mensagens são tratados isoladamente e não serão levados em consideração. *Note que o tamanho do vetor de mensagem e o termo de dados de cada pixel será reduzido a metade nesse passo.*
11	**Caso contrário**
12	Calcule a energia total de cada pixel e cada disparidade candidata segundo a Equação 2
13	**Fim Se**
14	**Fim Para**
15 :	O valor de disparidade que minimiza a energia individual de cada pixel e escolhido

3. OpenMP

OpenMP [9] foi desenvolvida para permitir a programação paralela em sistemas de memória compartilhada. Esta *Application Programming Interface* (API) provê suporte à paralelização de diversos tipos de aplicações. Além disto, foi criada com intuito de ser relativamente fácil de aprender e aplicar. OpenMP foi projetada para permitir uma abordagem incremental de paralelização a partir de um código seqüencial existente, possibilitando a paralelização de porções distintas de um programa.

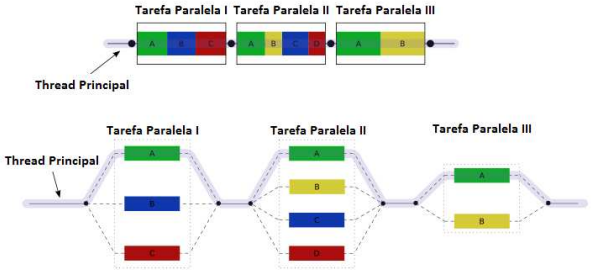

Figura 3: Exemplo da execução de programa paralelo em OpenMP.

A Figura 3 apresenta um exemplo do fluxo de um programa implementado de forma seqüencial e paralela em OpenMP. As tarefas I, II e III são partes do programa paralelizáveis. No programa seqüencial, a *thread* principal executa todas as tarefas. Por outro lado, no programa paralelo em OpenMP, a tarefa I é executada pela *thread* principal e mais duas outras, que são criadas rapidamente para o processamento desta tarefa. Quando a tarefa I termina, as *threads* criadas são destruídas e a *thread* principal continua a execução sequencial do programa até chegar numa nova tarefa paralelizável.

Com um modelo portátil e escalável, OpenMP fornece aos programadores uma interface simples e flexível para desenvolver aplicações paralelas para plataformas que vão desde *desktops* à supercomputadores. A API OpenMP oferece um ferramental que permite ao programador: criar grupos de *threads* para execução paralela; especificar o compartilhamento de trabalho entre as *threads* destes grupos; declarar variáveis compartilhadas e privadas;

sincronizar *threads* e permitir que elas também possam executar algumas operações exclusivamente (i.e, sem interferência de outras *threads*).

3.1. Implementação Paralela em OpenMP do algoritmo CSBP

A implementação paralela em OpenMP do algoritmo CSBP foi feita de forma incremental. Utilizou-se a ferramenta *g-prof (Gnu Profiler)* para a análise das funções mais custosas. Assim foi possível dar prioridade a tais funções com maior tempo de processamento. Porém essa abordagem de paralelização não foi suficiente para conseguir um *speed-up* significativo que garantisse uma maior vazão (*throughput*) de frames por segundo. Sendo assim, o código foi refeito como indicado no Algoritmo 2.

No momento da criação das *threads*, na linha 2 do Algoritmo 2, é iniciado o processo de paralelização do algoritmo. A cada tarefa principal do CSBP (representadas nas linhas 3, 5, 7, 8, 9 e 10 do Algoritmo 2), as *threads* são sincronizadas e distribuídas para execução paralela. Na tarefa da linha 3, as *threads* são responsáveis pela iteração nas linhas da imagem, calculando o termo de dados E_D para cada pixel no nível mais baixo (nível S - 1) e selecionando os k_{S-1} custos de dados que serão passados ao passo 5 em cada pixel. Para outras tarefas, o paralelismo segue o mesmo padrão, porém, em um nível de resolução espacial menor, como detalhado no pseudo código do Algoritmo 2.

A cada novo par de imagens recebido da câmera, é chamada uma função que representa o Algoritmo 2, que irá computar um novo mapa de disparidades. Como resultado do paralelismo em OpenMP, foi alcançada uma vazão de 13 FPS com uma imagem de 320x240 pixels. Contudo, uma câmera estéreo tradicional captura imagens a taxas superiores a 30 FPS [10] para resoluções como esta. Sendo assim, nossa versão paralela em OpenMP não foi suficiente para consumir toda entrada de dados da câmera. Por isso, uma nova abordagem de paralelização que fosse mais rápida foi implementada utilizando-se de GPUs CUDA.

Algoritmo 2: Constant-Space Belief Propagation OpenMP	
1	Inicialize as mensagens com zero
2	Criação das ***threads***
3	Cada ***thread*** interam nas linhas da imagem calculando o termo de dados E_D para cada pixel no nível mais baixo (nível $S - 1$) e seleciona os k_{S-1} custos de dados, que serão passados ao passo 5 em cada pixel. Os outros custos de dados e os níveis de disparidades correspondentes são tratados isoladamente.
4	**Para** $s = S - 1$ ate 0 **faça**
5	Para cada pixel X de nível mais grosso, atualize de forma paralela iterando sobre o vetor de mensagens para cada grupo de linhas em cada pixel X e para cada disparidade de acordo com a Equação 1.
6	**Se** s > 0 **então**
7	Inicialize as mensagens para o próximo nível, paralelizando os pixels X_j das colunas da imagem, utilizando a Equação 3 e selecione os níveis de disparidade.
8	Cada ***thread*** calcula o termo de dados para cada pixel X da imagem para o próximo nível, usando os níveis de disparidade selecionados.
9	Cada ***thread*** calcula a energia total de cada pixel X_j das colunas da imagem para cada disparidade candidata de acordo com a Equação 2.
10	Cada ***thread*** seleciona $k_{s-1} = k_s/2$ níveis de disparidade e valores de mensagem correspondente aos k_{s-1} menores valores de energia. Posteriormente de forma seqüencial aplique o passo numero **5** para cada pixel do nível atual. Os outros valores de disparidades e mensagens são tratados isoladamente e não serão levados em consideração.
11	**Caso contrário**
12	Calcule a energia total de cada pixel e cada disparidade candidata segundo a Equação 2
13	**Fim Se**
14	**Fim Para**
15 :	O valor de disparidade que minimiza a energia individual de cada pixel e escolhido

4. C+CUDA

Para o programador de C+CUDA, a GPU é um co-processador da CPU capaz de executar dezenas de milhares de *threads* em paralelo. Trechos da aplicação com grandes demandas de computação podem ser traduzidos para C+CUDA e executados em GPUs. Para traduzir estes trechos para C+CUDA, o programador tipicamente reescreve sua versão seqüencial na forma de *kernels* paralelos, que podem ser funções simples ou aninhamentos complexos de funções.

Um *kernel* comanda a execução na GPU de um conjunto de *threads*, que são organizadas em grades (*grids*) de blocos de *threads* (*thread blocks*). Um *grid* é um conjunto de *thread blocks* que executam independentemente, enquanto que um *thread block* é um conjunto de *threads* que podem cooperar por meio de sincronização do tipo barreira e acesso compartilhado a um espaço de memória exclusivo de cada *thread block*.

O grande número de *threads* por processador, centenas de processadores por GPU e o baixo custo do chaveamento entre *threads* (o que oculta a latência de memória) viabilizam o modelo de computação massivamente paralelo e de alto desempenho de C+CUDA.

4.1. Arquiteturas de GPUs CUDA

A Fermi é a nova geração de GPUs da Nvidia, desenvolvida primariamente para aplicações de propósito geral. Entre as principais melhorias desta geração em relação às anteriores estão: o aumento do desempenho em operações de ponto flutuante de precisão dupla, sendo oito vezes mais rápido do que a GT200; suporte a ECC (*Error-correcting Code*); e introdução de uma hierarquia de *cache* de dois níveis [11], permitindo uma maior velocidade em operações atômicas.

A arquitetura Fermi é o salto mais significativo na arquitetura de GPU desde o lançamento da G80 [11]. A G80 foi a visão inicial de uma arquitetura unificada de computação paralela de propósito geral e processamento gráfico. A GT200 estendeu o desempenho e as funcionalidades da G80.

Na GT200, a ALU inteira era limitada a 24 *bits* de precisão para operações de multiplicação, sendo necessária emulação para esta operação em 32 *bits*. Por outro lado, na Fermi, a ALU inteira suporta precisão de 32 bits para todas as instruções, de acordo com os requisitos das linguagens C/C++. A ALU inteira também é otimizada para suportar operações de 64 *bits*.

A primeira GPU da linha Fermi foi implementada com 3 bilhões de transistores, possuindo até 512 CUDA *cores*. Cada núcleo executa uma instrução de ponto flutuante ou inteiro por ciclo de *clock* para uma *thread*. Os 512 *cores* estão organizados em 16 SMs (*Streaming Multiprocessor*) com 32 *cores* cada. A GPU Fermi possui 6 bancos de memória com endereçamento de 64 bits, suportando até 6 GB de memória GDDR5 DRAM. A interconexão com o *host* (CPU) é feita através do barramento PCI-Express.

O SM cria, gerencia, escalona e executa grupos de 32 *threads* paralelas, denominados *warps*. *Threads* individuais de um mesmo *warp* iniciam a execução juntas em uma mesma instrução, entretanto possuem seus próprios registradores. Diferente das gerações anteriores a Fermi é capaz de escalonar e despachar dois *warps* simultaneamente em um mesmo SM. O duplo escalonador de *warps* seleciona dois *warps* e despacha uma instrução de cada *warp* para um grupo de 16 cores, 16 unidades de *load*/*store* ou 4 SFU (*Special Function Unit*).

4.2. Implementação Paralela em C+CUDA do Algoritmo CSBP

Após a implementação paralela do algoritmo em OpenMP já se conhecia de antemão as tarefas que demandavam mais tempo de processamento e que, portanto, eram candidatas a se tornarem funções de *kernel* na GPU. Ao invés de construir um único kernel englobando todas as tarefas, tomou-se a decisão de projeto de dividir as tarefas em *kernels* separados para garantir a disponibilidade de dados na memória global da GPU entre tarefas.

Usando uma abordagem iterativa, optou-se por paralelizar cada tarefa individualmente mantendo a cópia de dados entre a memória principal e a GPU na chamada do *kernel* e então fazer a cópia de volta para não afetar o fluxo subsequente das tarefas ainda não paralelizadas na GPU.

Para minimizar a latência da transferência de dados entre CPU e GPU, recomenda-se [12] que todas as tarefas (mesmo as pequenas em questão de ganho de desempenho) sejam transformadas em *kernels* para que o fluxo de processamento de dados na GPU seja contínuo sem que haja transferências intermediárias entre CPU e GPU.

Dessa forma, o algoritmo CSBP em C+CUDA receberá como entrada um par de imagens da câmera, que serão processadas na GPU, e devolverá o mapa de disparidade correspondente. Essas cópias estão exemplificadas nas linhas 2 e 14 do Algoritmo 3.

Considerando a estrutura piramidal proposta pelo algoritmo HBP, a relação entre número de disparidades e granularidade do pixel é variável em função do nível de disparidade L da pirâmide: à medida que se eleva o nível na pirâmide, a granularidade do pixel aumenta e a faixa de disparidade diminui. Fazendo a correspondência do número de disparidades com o número de *threads* e do número de pixels com o número de blocos do *grid*, verifica-se que há um *trade-off* a ser considerado na alocação de blocos e *threads* na GPU para cada nível da pirâmide de execução do algoritmo CSBP.

Nessa implementação CUDA, todos os blocos são bidimensionais para todos os *kernels* (exceto linhas 10 e 13) e foram assim dimensionados devido à necessidade de se percorrer a imagem ao longo da sua altura e largura. No caso das *threads* do *kernel* da linha 5, a escolha bidimensional deve-se ao relacionamento 4-conectado existente entre cada pixel (x,y) da imagem com seus vizinhos, fazendo com que seja necessária uma *thread* para cada pixel vizinho (x,y+1), (x,y-1), (x+1,y) e (x-1,y) para propagar as mensagens.

O algoritmo busca encontrar o mínimo de uma função de energia (disparidade), sendo necessário sincronizar esses valores entre as *threads*. Para tanto, foi utilizada uma operação atômica (atomicMin) na memória compartilhada de *threads* do mesmo bloco. A utilização dessa operação de barreira na memória compartilhada é possível e eficiente a partir da *Capability* 1.3 das placas CUDA NVIDIA.

Como resultado do paralelismo em C+CUDA, foi alcançada uma taxa de 66 FPS com imagens coloridas de resolução 320x240 pixels. Apesar de uma câmera estéreo tradicional capturar imagens a 48 FPS [10] com resolução de 640x480 pixels, nossa versão paralela em C+CUDA tem capacidade de processamento superior a captura da câmera e, portanto, adequada ao propósito

de aplicação de câmeras estéreo como um sensor tridimensional para navegação robótica em tempo real.

Algoritmo 3: Constant-Space Belief Propagation CUDA	
1	Inicialize as mensagens com zero na memória da GPU (cudaMemSet)
2	Copia o par de imagens para memória da GPU (cudaMemcpy)
3.1	Uma função *kernel* com *grid* bidimensional e bloco unidimensional onde cada *thread* do bloco corresponde a um termo de dados E_D no nível mais baixo (nível $S - 1$)
3.2	Ainda nessa função *kernel* foram selecionados os k_{S-1} custos de dados e passados ao passo 5 em cada pixel. Os outros custos de dados e os níveis de disparidades correspondentes são tratados isoladamente.
4	**Para** $s = S - 1$ ate 0 **faça**
5	Uma função *kernel* com *grid* bidimensional e blocos bidimensionais onde cada *thread* do bloco atualiza iterativamente o vetor de mensagens em cada pixel e para cada disparidade de acordo com a Equação 1.
6	**Se** s > 0 **então**
7	Uma função *kernel* com *grid* bidimensional e blocos bidimensionais onde cada *thread* do bloco corresponde a um termo de dados E_D para o próximo nível, usando os níveis de disparidade selecionados. O resultado é usado no passo 8.3.
8.1	Uma função *kernel* com *grid* bidimensional e bloco unidimensional onde somente a *thread* zero de cada bloco inicializa as mensagens para o próximo nível usando a Equação 3 e seleciona os níveis de disparidade.
8.2	Na mesma função de *kernel* todas as *threads* do bloco calculam a energia total de cada pixel para cada disparidade candidata de acordo com a Equação 2.
8.3	Ainda no mesmo *kernel* as *threads* zero de cada bloco selecionam $k_{s-1} = k_s/2$ níveis de disparidade e valores de mensagem correspondente aos k_{s-1} menores valores de energia e aplicam a estes o passo numero **5** para cada pixel. Os outros valores de disparidades e mensagens são tratados isoladamente e não serão levados em consideração.
9	**Caso contrário**
10	Uma função de *kernel* com grid e blocos unidimensionais em que as *threads* calculam a energia total de cada pixel e cada disparidade candidata segundo a Equação 2
11	**Fim Se**
12	**Fim Para**
13:	Uma função de *kernel* com grid e blocos unidimensionais em que as *threads* selecionam o valor de disparidade que minimiza a energia individual de cada pixel
14	Copia o mapa de disparidade da memória da GPU para a memória principal do computador

5. Metodologia

Nós implementamos uma versão seqüencial do algoritmo CSBP em C, uma paralela em OpenMP, e uma paralela em C+CUDA. Para avaliar o desempenho dos Algoritmos 1 e 2, os experimentos foram executados numa CPU *multi-core* e, para o Algoritmo 3, em duas GPUs distintas.

Para a aferição dos tempos de execução, foi utilizada a função *gettimeofday* da API padrão do Linux. Os resultados foram obtidos pela média de 5 execuções sucessivas do CSBP, mantendo-se a mesma resolução de imagem para uma mesma plataforma. Os tempos de execução medidos para CUDA levam em consideração o *overhead* de transferência de dados CPU-GPU.

Para avaliar o desempenho de cada versão paralela, foi utilizada a métrica *speed-up*, dada pela razão entre o tempo de execução da versão seqüencial e o tempo de execução da versão paralela para imagens de resoluções diferentes.

5.1. Base de Dados

A imagem *plant*, utilizada em nossos experimentos, faz parte do conjunto de teste *dataset2006* e foi obtida no trabalho apresentado em [4]. Para avaliar o desempenho do algoritmo paralelo, a imagem original de resolução 800x600 pixels foi re-escalada para resoluções de 320x240, 640x480 e 1024x768 pixels. Tais experimentos foram realizados com o propósito de avaliar em quais situações seria possível a execução do algoritmo CSBP em tempo real.

5.2. CPU e GPU

Os experimentos seqüenciais, em OpenMP e parte dos experimentos CUDA foram executados em uma máquina Dell Alienware, com CPU Intel Core i7 930 (*Quad Core* e *Hyper-threading*) de 2,8 GHz, 2MB de *cache* L3 por core e 12GB de DRAM DDR3 de 1333 MHz. A placa de vídeo utilizada nestes experimentos CUDA foi a NVIDIA GeForce GTX480, com 1,5 GB de DRAM GDDR5.

A GPU da GTX 480 possui 15 *Stream Multiprocessors* (SM), cada uma com 32 CUDA *cores* (ou *Stream Processors* - SPs) para executar operações inteiras e de ponto flutuante, totalizando 480 *cores*. Esta GPU permite criar 1024 *threads* por bloco de *thread*, manter o estado de 23K threads simultaneamente (um *grid* de 23K *threads*) e executar até 16 *kernels* concorrentes iniciados por um mesmo processo de CPU. Além disso, ela permite processar dados em precisão dupla com desempenho máximo teórico de 168 Gflop/s, e de precisão simples a 1,35 Tflop/s [11, 12].

Os demais experimentos em CUDA foram executados em uma máquina Dell Optiplex, com CPU AMD Athlon 64 X2 (*Dual Core*) 5.200+ de 2,7 GHz, 512KB de cache L2 por core e 3GB de DRAM DDR2 de 800 MHz. A placa de vídeo utilizada foi a NVIDIA GeForce GTX 285, com 1GB de DRAM GDDR3. Os SPs da GTX 285 são agrupados em SMs que, por sua vez, são agrupados em *Thread Processing Clusters*

(TPCs [13]). A GTX 285 possui 10 TPCs de 3 SMs, cada um com 8 SPs, totalizando 240 *cores*.

A GTX 285 permite criar 512 *threads* por bloco e manter o estado de 32K *threads* simultaneamente (um *grid* de 32K *threads*). Ela permite também processar dados de ponto flutuante em precisão simples e dupla. Entretanto, o desempenho máximo em precisão dupla é de 80 Gflop/s, enquanto que o em precisão simples é de 933 Gflop/s [13].

6. Resultados Experimentais

A Figura 4 apresenta os tempos de execução (em segundos) obtidos com as versões sequencial e paralelas em OpenMP e em C+CUDA em função de diferentes resoluções de imagens (320x240, 640x480, 800x600, e 1024x768 pixels). Para melhorar a visibilidade da diferença entre os resultados obtidos com a versão paralela em C+CUDA usando a GTX285 e a GTX480, re-apresentamos esses resultados na Figura 5.

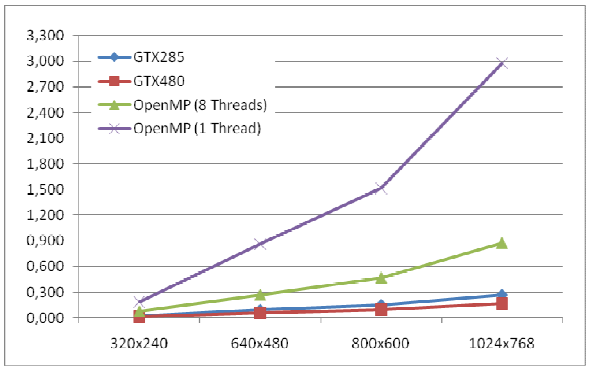

Figura 4: Tempos de execução em função da resolução da imagem.

Os gráficos da Figura 4 e da Figura 5 mostram que a versão paralela em C+CUDA usando a GTX285 apresentou menor tempo de execução comparado ao das versões seqüencial e paralela em OpenMP. Contudo, a versão paralela em C+CUDA usando a GTX480 foi a que apresentou o melhor resultado, pois esta nova geração de GPUs possui o dobro de *cores* de sua antecedente (GTX285) e uma hierarquia de *cache* de dois níveis.

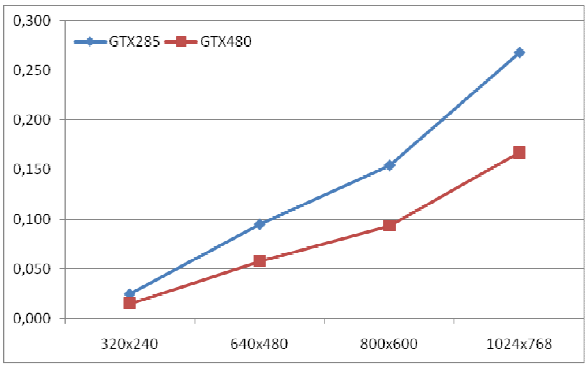

Figura 5: Tempos de execução em função da resolução da imagem usando a GTX285 e a GTX480.

A Figura 6 apresenta as taxas de FPS obtidas com as versões seqüencial e paralelas em OpenMP e C+CUDA em função de diferentes resoluções de imagens (320x240, 640x480, 800x600, e 1024x768 pixels). A taxa de FPS é dada pelo inverso do tempo de execução. A maior taxa de FPS – 66 FPS – foi obtida com a versão paralela em C+CUDA usando a GTX480 e processando imagens de resolução 320x240 pixels.

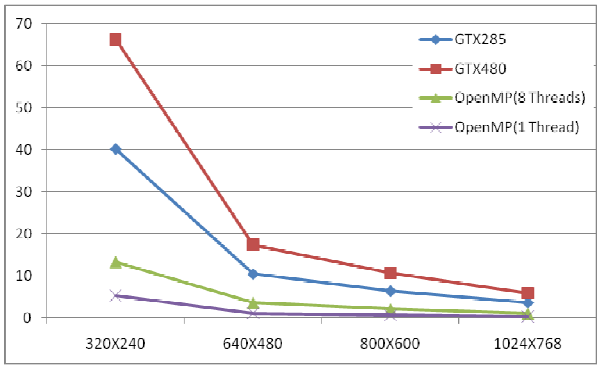

Figura 6: FPS em função da resolução da imagem.

A Figura 7 apresenta o *speed-up* obtido com as versões seqüencial e paralelas em OpenMP e C+CUDA em função de diferentes resoluções de imagens (320x240, 640x480, 800x600, e 1024x768 pixels). O maior *speed-up* – 18x – foi obtido com a versão paralela em C+CUDA usando a GTX480 e processando imagens de resolução 1024x768 pixels. Com esta resolução de imagem, a versão paralela em C+CUDA usando a GTX480 foi aproximadamente 5x mais rápida do que a versão paralela em OpenMP usando 8 *threads* e 1,6x mais rápida do que a versão em C+CUDA usando a GTX285.

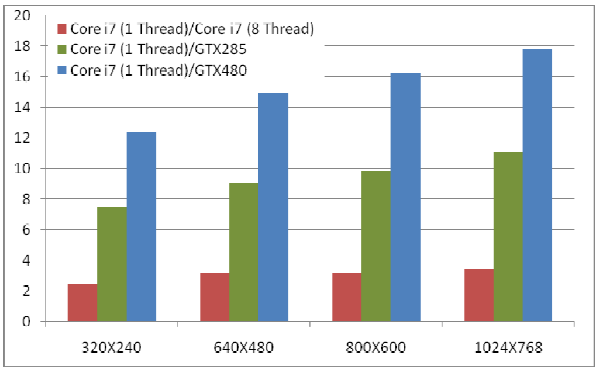

Figura 7: Speed-up em função da resolução da imagem.

A Tabela 1 sumariza o desempenho em termos de FPS obtido com as versões sequencial e paralelas em OpenMP e em C+CUDA em função de diferentes resoluções de imagens. Na Tabela 1, as linhas denotam a versão da implementação (seqüencial, paralela em OpenMP, paralela em C+CUDA usando a GTX285 e a GTX480) e as colunas denotam a resolução da imagem (320x240, 640x480, 800x600, e 1024x768 pixels).

Tabela 1: Desempenho em termos de FPS obtido com as versões seqüencial e paralelas.

	320x240	640x480	800x600	1024x768
Sequencial	5,36	1,16	0,66	0,34
OpenMP (8 *threads*)	13,33	3,70	2,13	1,14
C+CUDA (GTX285)	40,18	10,52	6,48	3,73
C+CUDA (GTX480)	66,29	17,34	10,71	5,97

Como os resultados da Tabela 1 mostram, o desempenho absoluto da versão paralela em C+CUDA em termos de FPS foi significativamente alto. Entretanto, o desempenho relativo em termos de *speedup* foi razoável em relação a trabalhos relacionados [3]. Uma provável explicação para o desempenho relativo obtido seria o baixo reuso de dados lidos da memória da GPU.

7. Conclusões

Neste artigo, desenvolvemos duas versões paralelas do algoritmo *Constant Space Belief Propagation* (CSBP) [1]: uma em OpenMP e outra em C+CUDA. Avaliamos o desempenho das nossas versões paralelas em C+CUDA e em OpenMP e da versão seqüencial em termos de tempo de execução, FPS e *speedup* usando imagens de diferentes resoluções (320x240, 640x480, 800x600, e 1024x768 pixels). Para imagens de 640x480 *pixels*, a versão sequencial teve desempenho de 1,16 FPS, a versão paralela em OpenMP teve desempenho de 3,7 FPS, enquanto que a versão paralela em C+CUDA teve desempenho de 17,3 FPS em máquinas desktop de alto desempenho. Esses resultados mostram que é possível empregar câmeras estéreo como sensores para navegação robótica em tempo real com imagens de até 640x480 pixels.

Uma direção para trabalho futuro seria investigar implementações paralelas de outros algoritmos de visão estéreo que apresentem maior paralelismo em nível de dados.

8. Referências

[1] Q. Yang, L. Wang, N. Ahuja. “A Constant-Space Belief Propagation Algorithm for Stereo Matching”, IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2010.

[2] S. Thrun, W. Burgard, D. Fox. “Probabilistic Robotics”, MIT Press, 2005.

[3] C.A. Carvalho, L.P. Veronessi, H. Oliveira, A.F. De Souza. “Implementation of a Biologically Inspired Stereoscopic Vision Model in C+CUDA”, accepted in GPU Tecnology Conference, Sep 30, 2009.

[4] Middlebury Stereo Vision Page – http://vision.middlebury.edu/stereo/, consultado em 05/08/2011

[5] W.T. Freeman, E. Pasztor, and O.T. Carmichael. “Learning low-level vision”, IJCV, 40(1):25–47, 2000.

[6] J. Sun, N. Zheng, and H.Y. Shum. “Stereo matching using belief propagation”, PAMI, 25(7):787–800, 2003.

[7] Q. Yang, C. Engels, and A. Akbarzadeh. “Near real-time stereo for weakly-textured scenes”, In BMVC, pages 80–87, 2008

[8] T. Yu, R.S. Lin, B. S., B. Tang. “Efficient message representations for belief propagation”. In ICCV , pages 1–8, 2007.

[9] B. Chapman, G. Jost, R.V. der Pas, David J. Kuck. “Using OpenMP: Portable Shared Memory Parallel Programming”. Scientific and Engineering Computation Series, The MIT Press. 2008.

[10] Point Grey – Stereo Vision – Bumblebee2 CCD FireWire camera , http://www.ptgrey.com, consultado em 05/08/2011.

[11] NVIDIA, “NVIDIA’s Next Generation CUDA Compute Architecture: Fermi”, 2009

[12] NVIDIA, “NVIDIA CUDA: Compute Unified Device Architecture - Programming Guide 4.0”, 2011.

[13] NVIDIA, “Technical Brief: NVIDIA GeForce GTX200 GPU Architectural Overview”, 2008.