Processamento de Consultas XPath em GPU

Dilson A. Guimarães, Filipe de L. Arcanjo,
Laura R. Antuña, Mirella M. Moro, Renato C. Ferreira

Universidade Federal de Minas Gerais, Brasil
{dilsonag, filipe, laura.antuna, mirella, renato}@dcc.ufmg.br

Abstract. Technologies such as CUDA and OpenCL have been have been making the usage of graphics cards (GPUs) for general purpose programming more popular, which provides impressive performance gains. However, using such cards for speeding up XML Databases is yet to be fully explored. XML databases offer much flexibility for Web-oriented systems. Nonetheless, such flexibility comes at a considerable computational cost. This work shows how graphics cards can be leveraged to reduce the computational cost of processing some of those queries. It presents an algorithm designed to consider the cost model of GPUs and to perform queries efficiently. An experimental study reveals that this algorithm is more efficient than implementations of a similar strategy on a CPU for all the datasets tested. The speedups with respect to exist-db, a popular XML database system, are as high as two orders of magnitude.

Resumo. Tecnologias como CUDA e OpenCL tem popularizado o uso de placas gráficas (GPUs) em aplicações de propósito geral, com ganhos de desempenho surpreendentes. No entanto, o uso dessas placas para melhorar o desempenho de Bancos de Dados XML ainda é pouco explorado. Esses Bancos de Dados oferecem grande flexibilidade para sistemas Web. Tal flexibilidade, entretanto, vem acompanhada de considerável custo computacional. Esse trabalho mostra como as placas gráficas podem ser utilizadas para reduzir o custo de algumas consultas feitas a esses sistemas. Ele apresenta um algoritmo que considera o modelo de custos das GPUs para processar consultas nessas placas eficientemente. Uma análise experimental revela que esse algoritmo é mais eficiente que implementações da mesma estratégia em CPUs em todas as bases de dados avaliadas. Os ganhos de performance com relação ao exist-db, popular sistema de bancos de dados, chegam a cerca de duas ordens de magnitude.

Categories and Subject Descriptors: I.3 Computer Graphics [**I.3.1 Hardware Architecture**]: Graphics processors; H.2 Database Management [**H.2.4 Systems**]: Query Processing

Keywords: Consultas XPath, XML, GPU

1. INTRODUÇÃO

O desenvolvimento de tecnologias como CUDA e OpenCL deu início a uma nova era na Computação. Graças a elas, o poder de processamento massivamente paralelo das placas gráficas (GPUs), antes restrito a domínios como renderização de imagens tridimensionais e jogos, foi disponibilizado aos programadores para uso em aplicações de propósito geral. Em consequência dessa abertura, centenas de problemas de diversos domínios ganharam soluções que, frequentemente, superam em muito o desempenho das implementações tradicionais, que não foram aceleradas por essas placas.

Programar as GPUs ainda é difícil e exige dos programadores uma ampla gama de conhecimentos. Essas placas possuem, tipicamente, até seis tipos diferentes de memória [Bakkum and Skadron 2010] e um modelo de custos bastante complexo. Esse modelo inclui características incomuns, tais como a ausência de chamadas recursivas e a presença de um fenômeno conhecido como *divergência*, que pode fazer com que parte do paralelismo seja perdida [Coutinho et al. 2011]. Face a essas dificuldades, os ganhos obtidos em áreas tais como Bioinformática, Simulações Científicas e Modelagem Estatística são, de fato, impressionantes. Muitas áreas, no entanto, permanecem pouco exploradas.

Este trabalho foi parcialmente financiado por CNPq, Fapemig e InWeb, Brasil.

É o caso dos Sistemas de Bancos de Dados e, especificamente, dos Bancos de Dados XML. Esses sistemas são populares em consequência do modelo de dados flexível, portável e hierárquico fornecido pela linguagem XML, que é especialmente adequado para aplicações web[1]. Porém, uma consequência indesejada desse modelo de dados rico é o fato de que, frequentemente, as consultas são computacionalmente caras. Essas consultas são especificadas por meio de uma linguagem própria, como por exemplo a XPath[2]. XPath modela os documentos XML como árvores de nós. Uma consulta XPath é uma forma de especificar um ou mais caminhos nessa árvore por meio de uma sintaxe compacta. Muitas consultas em XPath requerem que toda a árvore seja percorrida, o que eleva o seu custo computacional.

Esse trabalho mostra que o grande poder de processamento das GPUs pode ser a chave para reduzir os tempos de processamento das consultas a bases de dados em XML. Ele apresenta um processador de consultas projetado para executar no complexo hardware da placa gráfica e capaz de lidar com um subconjunto de XPath[3], explicitado na Seção 3. As contribuições são:

—Um algoritmo para consultas XML adequado ao modelo de custos das GPUs (Seção 3.1).
—Uma descrição de como esse algoritmo pode ser implementado nessas placas (Seção 3.2).
—Um estudo experimental desse algoritmo, que fornece indícios de que as GPUs são uma alternativa viável para as abordagens tradicionais de processamento de consultas XPath (Seção 4).

O processador de consultas proposto, conforme revelam os nossos testes, supera em desempenho implementações sequenciais e paralelas em CPU, bem como o conhecido sistema exist-db [eXist db 2012]. Os ganhos em relação à esse último chegam a duas ordens de magnitude (138 vezes).

2. CONCEITOS E TRABALHOS RELACIONADOS

A principal utilização de GPUs na área de Bancos de Dados relaciona-se às operações mais comuns como junção, divisão e agregação. Abordagens de integração fornecidas pela *Oracle* foram analisadas para que a implementação paralela em GPU de operações espaciais fossem integradas em bancos de dados comerciais, como procedimento externo [Bandi et al. 2004]. Foi também proposta a implementação paralela em GPU de operações como predicados, combinações booleanas e agregações sobre bancos de dados [Govindaraju et al. 2005], assim como algoritmos de junção a partir de primitivas como divisão e ordenação [He et al. 2008]. Utilizando a nova geração de GPUs NVidia, em que GPU e CPU compartilham um espaço de memória, [Kaldewey et al. 2012] obtiveram melhoras na eficiência de operações de junção em bancos de dados, com resultados superiores ao de [Kim et al. 2009].

Algoritmos de pesquisa e ordenação em bancos de dados também constituem uma área explorada por meio do uso de GPUs. [He et al. 2009] propõem um sistema de processamento de consultas relacionais em GPU, por meio da alta paralelização de primitivas como divisão e ordenação. A pesquisa por índice, estruturada em árvore, também é paralelizada por [Kim et al. 2010], e [Govindaraju et al. 2006] apresentam o algoritmo *GPUTeraSort*, que utiliza em conjunto CPU e GPU para ordenar bancos de dados que apresentam chaves na ordem de bilhões.

Implementações que lidam com um subconjunto de operações já conhecidas também são comuns. Tanto [Lin and Yuan 2011] quanto [Bakkum and Skadron 2010] propõem uma implementação direta na GPU de um subconjunto de comandos *SQLite*, com o foco na operação `SELECT` por parte do segundo. A interface genérica apresentada facilita a utilização dos usuários, que para manipular os bancos de dados de forma eficiente não precisam utilizar linguagens como CUDA ou modificar

[1]XML: `http://www.w3.org/TR/REC-xml/`
[2]XPath: `http://www.w3.org/TR/xpath/`
[3]É importante notar que trabalhos relacionados ao processamento de consultas XML mostram que tal subconjunto é reponsável por grande parte do desempenho de avaliação das mesmas.[Moro et.al 2009].

programas para utilizar bibliotecas diferentes das já existentes. Essa característica também pode ser observada em nosso trabalho, que implementa um subconjuto de operações de uma linguagem existente, com interface flexível. Além disso, nosso trabalho avança o estado-da-arte de SGBDs em GPUs por considerar o processamento de dados XML.

3. XPATH EM GPU

Antes de apresentar o algoritmo e sua implementação, esta seção discute alguns conceitos fundamentais para a compreensão dos mesmos. O subconjunto de XPath considerado foca em consultas estruturais, por essas terem grande influência no processamento geral das consulta XML [Moro et.al 2009]. Ele consiste nos operadores de filiação (/) e de descendência própria (//). Por simplicidade, esse subconjunto de XPath é nomeado EXPath. Assim como em XPath, consultas em EXPath são escritas como expressões de caminho, conforme a gramática a seguir, onde o não terminal $< tag >$ é capaz de gerar qualquer *tag* XML válida:

$$< consulta > \rightarrow \; / < tag >< consulta > \; | \; // < tag >< consulta > \; | \; / < tag > \; | \; // < tag >$$

A estrutura de um documento XML pode ser representada como uma árvore rotulada enraizada. Cada vértice da árvore representa um elemento do documento XML e é rotulado com a *tag* do elemento. O elemento raiz do documento XML corresponde ao vértice raiz da árvore. A hierarquia dos elementos é representada por meio da relação de descendência na árvore.

Definição 3.1. *Árvore estrutural:* A árvore estrutural $T(d)$ de um documento XML d é uma tupla formada por uma árvore G, um vértice raiz $r \in V(G)$ e uma rotulação de G, $f : V(G) \rightarrow \sum$, em que $\sum$ é o conjunto de *tags* em d. O conjunto de vértices de G é o conjunto de elementos de d. Existe uma aresta (u, v) em G se e somente se v estiver dentro e um nível abaixo de u em d.

Definição 3.2. *Caminho na árvore estrutural:* Um caminho para um vértice v na árvore estrutural $T(d)$ de um documento XML d, $P(T(d), v)$, é uma sequência de vértices $(v_1, v_2, \ldots, v_n)$ tal que $v_1 = r(T(d))$, $v_n = v$ e, para todo par (v_i, v_{i+1}), $(v_i, v_{i+1}) \in E(G(T(d)))$. Define-se $r(T(d))$ como sendo a raiz da árvore estrutural de d e $G(T(d))$ como o grafo da arvore estrutural de d.

Definição 3.3. *Rotulação de caminho:* Seja f a rotulação da árvore estrutural $T(d)$ de um documento XML d, a rotulação $R(p)$ de um caminho $p = P(T(d), v)$ é a sequência obtida a partir do mapeamento de cada elemento de p em seu rótulo por meio de f:

$$R(p) = (f(v_1), f(v_2), \ldots, f(v_n)) \qquad v_i \in p$$

Toda expressão de caminho pode ser traduzida para uma expressão regular. Para isso, basta substituir as ocorrências do operador // por $.^*$ e eliminar as ocorrências do operador /. A expressão $.^*$ denota a ocorrência de qualquer símbolo zero ou mais vezes. A expressão de caminho /a//b pode ser traduzida para $a.^*b$, por exemplo. A expressão regular $e(q)$ obtida a partir de uma expressão de caminho q é chamada expressão regular induzida por q. Define-se $L(e(q))$ como o conjunto de todas as palavras reconhecidas por $e(q)$.

A resposta de uma consulta em EXPath para um documento XML pode ser definida como o conjunto de vértices finais dos caminhos da árvore estrutural do documento cuja a rotulação pode ser reconhecida pela expressão regular induzida pela consulta.

Definição 3.4. *Conjunto resposta:* O conjunto resposta $Q(d, q)$ de uma consulta q em um documento d é definido como o conjunto de vértices v tais que $R(P(T(d), v)) \in L(e(q))$.

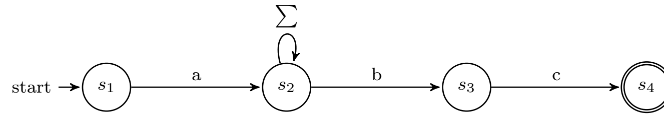

Fig. 1. Autômato finito não determinístico induzido pela consulta /a//b/c.

3.1 Algoritmo

É possível construir um autômato finito não determinístico (AFND) que reconheça a mesma linguagem que uma expressão regular induzida por uma consulta. Dada uma consulta com n operadores, é possível construir um AFND com $n+1$ estados. Um AFND que reconhece a mesma linguagem que a expressão regular induzida por uma consulta é chamado de AFND induzido pela consulta. A consulta /a//b/c, por exemplo, induz a expressão regular a.*bc e o AFND da Figura 1, onde $\sum$ denota todos os símbolos possíveis.

Uma maneira de executar uma consulta em um documento é simular uma instância do AFND induzido pela consulta para cada vértice da árvore estrutural do documento, considerando como entrada de cada instância a rotulação do caminho para o respectivo vértice. Os vértices para os quais a rotulação do caminho for reconhecida formam o conjunto resposta.

Caminhos na árvore estrutural apresentam sobreposição. Em outras palavras, os caminhos para quaisquer dois vértices compartilham um prefixo. Por exemplo, se o vértice u é descendente do vértice v, o caminho para v é prefixo do caminho para u. Da mesma forma, se o vértice a é pai dos vértices b e c, o caminho para a é prefixo dos caminhos para b e para c. Essa sobreposição é explorada no algoritmo elaborado para reduzir o custo de simular as várias instâncias do AFND induzido.

O algoritmo desenvolvido para o processamento de consulta em um documento consiste em realizar uma busca em largura na árvore estrutural do documento simulando várias instâncias do AFND induzido pela consulta. Existe uma instância associada a cada vértice. Inicialmente, a instância associada à raiz se encontra no estado inicial. Quando um vértice é visitado, a instância do AFND associada a esse vértice tem seu conjunto de estados ativos alterado para que seja igual ao conjunto de estados ativos da instância associada ao vértice pai. Em seguida, o rótulo do vértice é fornecido como símbolo de entrada para instância do AFND associada ao vértice. Se o estado final é atingido, o vértice é adicionado ao conjunto resposta.

O pseudocódigo desenvolvido é apresentado no Algoritmo 1. É mantido um conjunto de estados ativos S_v do AFND induzido pela consulta para cada vértice v da árvore estrutural do documento. O predicado $Fim\left(S_v\right)$ é verdeiro se e somente se o estado final do AFND induzido estiver em S_v. Transições são realizadas por $Próximo(S_v, a)$, que é o conjunto de estados ativos após o símbolo a ser fornecido como entrada para o AFND induzido com o conjunto de estados ativos S_v. O vértice pai de um vértice v é dado por $Pai\left(v\right)$. O nível de um vértice é definido como a sua distância da raiz em número de arestas.

No Algoritmo 1, cada vértice é visitado uma única vez. As operações envolvendo o conjunto de estados ativos S_v em um vértice podem ser realizadas em tempo linear em $|S_v|$. Como o número de vértices é equivalente ao número de elementos no documento XML $|d|$, e número de estados no AFND induzido é uma unidade maior do que o tamanho da consulta em número de operadores $|q|$, a ordem de complexidade de tempo do Algoritmo 1 é $O\left(|d|\,|q|\right)$.

O laço interno do Algoritmo 1 pode ser executado de forma paralela. Para isso, é preciso que o acesso ao conjunto resposta seja sincronizado. Considerando um número polinomial de processadores, uma consulta q em um documento d pode ser processada em tempo $O\left(h\left(T\left(d\right)\right)|q| + |Q\left(d,q\right)|\right)$, onde $h\left(T\left(d\right)\right)$ é a altura da árvore estrutural do documento d.

Entrada: Uma consulta q e um documento d
Saída: Conjunto de elementos em d que satisfazem q
$Q \leftarrow \emptyset$;
$S_{r(T(d))} \leftarrow$ *Próximo* $(\emptyset, f(r(T(d))))$;
se $Fim\left(S_{r(T(d))}\right)$ **então**
  $Q \leftarrow Q \cup \{r(T(d))\}$;
fim
para cada *nível i em T (d) exceto o nível da raiz* **faça**
  para cada *vértice v no nível i de T (d)* **faça**
    $S_v \leftarrow$ *Próximo* $\left(S_{Pai(v)}, f(v)\right)$;
    se $Fim(S_v)$ **então**
      $Q \leftarrow Q \cup \{v\}$;
    fim
  fim
fim
retorna Q

Algoritmo 1: Algoritmo de processamento de consultas.

3.2 Implementação

A estratégia descrita na seção anterior foi implementada para executar sobre uma representação da árvore estrutural do documento XML que é mantida na memória da placa gráfica. Essa representação é obtida à partir do documento por meio de duas etapas: análise e serialização. Essas etapas são mostradas na figura 2, que contém uma visão geral da arquitetura implementada.

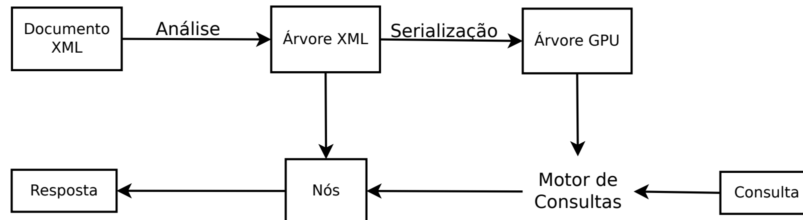

Fig. 2. Visão geral da arquitetura do processador de consultas

A função da análise é converter o documento XML, inicialmente em formato texto, em uma árvore que é armazenada na memória principal do computador. A leitura do documento é feita por meio da biblioteca xerces[4]. Diferentemente da árvore estrutural, definida na seção anterior, essa árvore contém informação o suficiente para reconstruir todo o XML. Isso é necessário para que as consultas possam ser respondidas de maneira legível uma vez que os resultados retornem da GPU.

A etapa de serialização é responsável por converter a árvore obtida pela análise numa representação serializada, em que os vértice são armazenados de maneira contígua na memória. Essa representação é o equivalente em código à árvore estrutural apresentada na definição 3.1. Cada vértice na árvore serializada contém três inteiros que representam (i) a *tag* do vértice, (ii) a posição do vértice-pai na árvore serializada e (iii) o conjunto de estados ativos – representados por bits – do AFND após receber o rótulo do vértice como entrada. Essa representação compacta é possível porque o algoritmo 1 não requer nenhuma informação além desses três números para executar, e é interessante pois as placas gráficas disponíveis atualmente possuem pouca memória[5]. Os vértice são armazenados na árvore serializada na mesma ordem em que apareceriam se a árvore estrutural fosse percorrida por meio de

[4] Xerces: `http://xerces.apache.org/xerces-c/`
[5] A *GeForce GTX 470* usada nos experimentos, por exemplo, tem apenas 1.25GB

uma busca em largura. Isso é feito porque essa representação aumenta a localidade espacial dos dados, aspecto importante a ser considerado já que as GPUs leem da memória em grandes blocos.

O processador de consultas propriamente dito é implementado utilizado a plataforma CUDA, da *nvidia*, por meio de um único *kernel*. No vocabulário de CUDA, um kernel é um procedimento que executa diretamente no hardware da placa gráfica em paralelo. Esse kernel é chamado uma vez para cada nível da árvore estrutural. Cada uma dessas chamadas dispara uma thread para cada vértice no nível. Cada thread obtém o conjunto de estados que estavam ativos anteriormente a partir do identificador do vértice-pai e executa as transições apropriadas do AFND com base na tag do vértice. O comportamento é equivalente ao conteúdo do loop mais interno do algoritmo 1.

A implementação desse kernel é cuidadosa para evitar divergências. Ela utiliza operações de bits para simular as transições do AFND, o que elimina muitos dos desvios condicionais que seriam necessários caso isso não fosse feito. Uma divergência ocorre quando uma das threads toma um caminho de execução diferente das demais ao encontrar um desvio. Nessa situação, o hardware da placa gráfica serializa parcialmente a execução. Ele executa todas as threads que tomaram um caminho no desvio antes de executar as que tomaram o outro.

As threads reportam os resultados para uma região de memória centralizada, que armazena identificadores dos vértices nos quais houve reconhecimento. O acesso a essa região é sincronizado por meio de uma rotina de adição atômica fornecida pela plataforma CUDA[6]. A presença dessa rotina e o teste de reconhecimento são, provavelmente, os dois maiores gargalos de performance da implementação. Isso ocorre porque o teste envolve um desvio condicional potencialmente divergente [Coutinho et al. 2011] e alguma sincronização para que a adição atômica funcione.

Após o processamento da consulta, os identificadores de vértices encontrados são transportados de volta para a memória principal do computador. Esses identificadores são pareados com os respectivos vértices da árvore gerada pela análise para que os resultados possam ser apresentados para o usuário.

4. AVALIAÇÃO EXPERIMENTAL

Para avaliar o desempenho do motor de consultas implementado foram realizadas consultas em bancos de dados utilizando diferentes implementações do algoritmo, de modo que comparações dos tempos de execução obtidos pudessem ser realizadas. Especificamente, duas versões do algoritmo de busca na árvore XML foram implementadas, de forma a serem comparadas com a implementação em CUDA. Na primeira o algoritmo é paralelizado em CPU, por meio da plataforma OpenMP[7], enquanto a segunda consiste em uma versão serial do processador de consultas. Também foi utilizado para comparação o eXist[8], um SGBD popular XML nativo. A execução dos testes foi feita em um servidor com processador *Intel Xeon E5620*, 32GB de memória principal e placa gráfica *nVidia GeForce GTX 470* com 1.25GB de memória. Como o eXist é programado para utilizar a memória secundária, os testes realizados com ele foram executados diretamente de um *ramdisk*.

4.1 Bases de Dados

Com objetivo de avaliar o processador de consultas nos mais variados ambientes, consultas foram feitas em quatro bases de dados de tamanhos diferentes, variando entre 24MB e 1GB. Informações sobre cada uma das bases podem ser encontradas na Tabela I[9]. A coluna *profundidade máxima* indica a altura da árvore formada pelo documento XML, enquanto a *profundidade média* corresponde à profundidade média das folhas dessa árvore. Para cada base, foram realizadas 4 consultas diferentes, cada uma

[6] atomicAdd()
[7] Plataforma OpenMP: `http://www.openmp.org/`
[8] Banco de Dados eXist: `http://www.exist-db.org/`
[9] XML Data Repository: `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`

fazendo referência a nós encontrados em níveis e posições variadas da árvore XML, abrangendo assim a maioria dos casos de testes possíveis.

Tabela I. Datasets utilizados para testes

Dataset	Tamanho	Elementos	Atributos	Profundidade Máxima	Profundidade Média
dblp 1	1.1G	25.832.523	6.538.402	6	2,877659
psd7003	684MB	21.305.818	1.290.647	7	5,15147
dblp 2	128MB	3.332.130	404.276	6	2,90228
treebank	83MB	2.437.666	1	36	7,87279
nasa	24MB	476.646	56.317	8	5,58314

4.2 Resultados

A Tabela II mostra os tempos de execução resultantes das consultas realizadas, assim como o *speedup* obtido pela implementação em CUDA, comparada com as outras. Os resultados nos mostram que a implementação paralela em GPU é significativamente mais eficiente em todos os casos testados quando comparada com a versão em OpenMP, que por sua vez supera o desempenho da versão serial. Os testes realizados utilizando o *eXist* também demonstraram menor eficiência do que a implementação em GPU, principalmente para arquivos grandes, em que o speedup atingiu o valor máximo de 138. A Figura 3 ilustra esses resultados. Ela contém o speedup médio obtido com cada implementação.

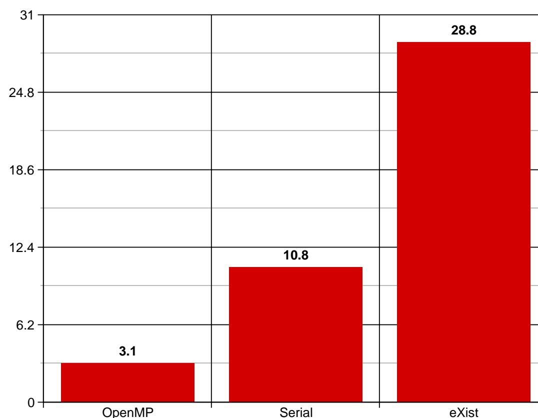

Fig. 3. Speedup Médio

Também é importante destacar a relação entre o tamanho do dataset utilizado e o desempenho obtido. Na implementação em CUDA o aumento do tamanho do arquivo representou um crescimento pequeno no tempo de execução, enquanto nos testes com as outras versões esse fator foi impactante no desempenho da consulta, principalmente no caso do *eXist*.

É importante notar que, embora ganhos de performance tenham sido obtidos em todas as bases de dados avaliadas, a solução apresentada nesse trabalho é limitada pela quantidade de memória disponível na GPU. Isso inviabiliza o uso do algoritmo proposto em bases de dados muito grandes. Nesses casos, soluções projetadas para realizar consultas na memória principal da CPU, em disco rígido ou mesmo soluções híbridas entre GPU e CPU poderiam ser mais adequadas.

5. CONCLUSÃO

Este trabalho mostrou como o poder de processamento das modernas placas gráficas pode amenizar um dos maiores gargalos dos SGBDs XML: o processamento de consultas estruturais. Nossa solução consiste em um algoritmo projetado especificamente para o modelo de custos dessas placas. Ele resolve as consultas decompondo-as em consultas menores que podem ser processadas em paralelo e adota uma representação compacta da base de dados, através de mapas de bits para maximizar a quantidade de informação que pode ser mantida na memória da placa e minimizar a quantidade de

Tabela II. Medidas de desempenho com base no tamanho do arquivo

Base	Consulta	#Elems.	Tempo de Execução (seg)				Speedup		
			CUDA	OMP	Serial	eXist	OMP	Serial	eXist
dblp 1	//author	6.563.296	0,114	0,377	0,354	8,298	3,3	3,1	75,3
	//article/author	2.156.500	0,047	0,193	0,390	6,525	4,1	8,3	138,8
	/dblp//article	844.543	0,027	0,119	0,300	0,515	4,4	11,1	19,1
	/author	0	0,013	0,070	0,291	0,869	5,4	22,4	66,8
psd7003	//authors	314.763	0,017	0,082	0,268	0,148	4,8	15,8	8,7
	//accinfo/db	1.199.979	0,031	0,124	0,354	0,415	4,0	11,4	13,4
	//db	1.531.327	0,036	0,133	0,269	1,376	3,7	7,5	38,2
	/ProteinDatabase/ProteinEntry	262.525	0,017	0,086	0,312	0,117	5,1	18,3	6,9
dblp 2	//title	328.859	0,007	0,019	0,042	0,139	2,7	6,0	19,8
	/dblp//volume	114.205	0,004	0,013	0,047	0,062	3,3	11,8	15,5
	//www/title	38	0,002	0,008	0,047	0,044	4,0	23,5	22,2
	/dblp/www	38	0,002	0,008	0,046	0,003	4,0	23,0	1,3
treebank	//NN	186.597	0,005	0,011	0,030	0,108	2,2	6,0	21,5
	//S//NP	419.255	0,009	0,019	0,042	0,278	2,1	4,7	30,9
	//S/NP	125.997	0,004	0,008	0,039	0,185	2,0	9,8	46,3
	/FILE/EMPTY	52.851	0,003	0,007	0,034	0,026	2,3	11,3	8,6
nasa	//name	71.688	0,002	0,003	0,006	0,033	1,5	3,0	16,8
	//other/name	286	0,001	0,001	0,007	0,016	1,0	7,0	15,7
	//year	5.935	0,001	0,001	0,006	0,006	1,0	6,0	5,8
	/datasets/dataset	2.435	0,001	0,001	0,007	0,004	1,0	7,0	4,3

desvios condicionais. Embora simples, nossa solução obteve performance superior ou equivalente a dos baselines em todas as cinco bases de dados consideradas, num total de vinte consultas XPath. O speedup obtido em comparação ao popular exist-db passou de duas ordens de magnitude (138 vezes). No futuro, pretende-se generalizar o algoritmo proposto para que consultas envolvendo predicados também possam ser realizadas nessas placas.

REFERÊNCIAS

Bakkum, P. and Skadron, K. Accelerating sql database operations on a gpu with cuda. In *Procs. of GPGPU*. Pittsburgh, Pennsylvania, pp. 94–103, 2010.

Bandi, N., Sun, C., Agrawal, D., and El Abbadi, A. Hardware acceleration in commercial databases: a case study of spatial operations. In *Procs. of VLDB*. Toronto, Canada, pp. 1021–1032, 2004.

Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Meira Jr., W. Divergence analysis and optimizations. In *Procs. of PACT*. Washington, DC, USA, pp. 320–329, 2011.

eXist db. exist-db open source native xml database. `http://exist-db.org/exist/index.xml`, 2012.

Govindaraju, N., Gray, J., Kumar, R., and Manocha, D. Gputerasort: high performance graphics co-processor sorting for large database management. In *Procs. of SIGMOD Conference*. Chicago, IL, USA, pp. 325–336, 2006.

Govindaraju, N. K., Lloyd, B., Wang, W., Lin, M., and Manocha, D. Fast computation of database operations using graphics processors. In *Procs. of SIGGRAPH*. Los Angeles, CA, 2005.

He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N. K., Luo, Q., and Sander, P. V. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.* 34 (4): 21:1–21:39, Dec., 2009.

He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., and Sander, P. Relational joins on graphics processors. In *Procs. of SIGMOD Conference*. Vancouver, Canada, pp. 511–524, 2008.

Kaldewey, T., Lohman, G., Mueller, R., and Volk, P. Gpu join processing revisited. In *Procs. of DaMoN*. Scottsdale, Arizona, pp. 55–62, 2012.

Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A. D., Kaldewey, T., Lee, V. W., Brandt, S. A., and Dubey, P. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Procs. of SIGMOD Conference*. Indianapolis, Indiana, USA, pp. 339–350, 2010.

Kim, C., Kaldewey, T., Lee, V. W., Sedlar, E., Nguyen, A. D., Satish, N., Chhugani, J., Di Blas, A., and Dubey, P. Sort vs. hash revisited: fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.* 2 (2): 1378–1389, Aug., 2009.

Lin, C.-F. and Yuan, S.-M. The design and evaluation of gpu based memory database. In *Procs. of ICGEC*. pp. 224 –231, 2011.

Moro et.al, M. M. XML: some papers in a haystack. *SIGMOD Record* 38 (2): 29–34, 2009.