Uma Abordagem para Processamento em Memória de Operações de Seleção em Sistemas de Bancos de Dados

Diego G. Tome [1], Marco A. Z. Alves[1], Eduardo C. de Almeida[1]

[1]Programa de Pós-Graduação em Informática (PPGINF)
Universidade Federal do Paraná (UFPR) – Curitiba – PR – Brasil

{dgtome,mazalves,eduardo}@inf.ufpr.br

Nível: Mestrado **Data de admissão:** 02/2016
Data de defesa da proposta: 05/2017 **Data esperada para defesa:** 02/2018
Passos concluídos:

– Implementação das estratégias tuple/column/vector-at-a-time.
– Resultados dos cenários x86/HMC.
– Publicação: Operand size reconfiguration for big data processing in memory. Design Automation and Test in Europe (DATE), 2017.

Próximos passos:

– Implementação da novas melhorias arquiteturais.
– Resultados da proposta final.
– Escrita da dissertação de mestrado.

Abstract. *A considerable portion of the time spent during databases operation processing consists in moving data around the memory hierarchy rather than actually processing it. The emergence of smart memories, as the new Hybrid Memory Cube (HMC) allows mitigating this memory wall problem, by executing instructions directly inside the memory, reducing data movement. In this work, we discuss the processing of databases inside the HMC. We focus on the select scan operator, because the scanning of columns moves large amounts of data prior to other costly operations like joins (i.e., push-down optimization). Our preliminary results with the tuple / column / vector-at-a-time query engines show performance gains of 30% for tuple-at-a-time, 11% column-at-a-time and 6× in the vector-at-a-time execution when compared to x86.*

Resumo. *Uma considerável porção do tempo gasto no processamento de operações de banco de dados consiste na movimentação de dados pela hierarquia de memória ao invés de um real processamento. O surgimento de memórias inteligentes, como o novo Cubo de Memória Híbrido (HMC) permite mitigar esse problema de "memory wall", executando instruções diretamente dentro da memória, reduzindo a movimentação de dados. Neste trabalho, é discutido o processamento de banco de dados dentro do HMC. O foco foi no operador de seleção, já que esta operação move grandes quantidades de dados antes de outras operações dispendiosas, como junções (ou seja, otimização push-down). Os resultados preliminares com mecanismos de consulta tuple/column/vector-at-a-time apresentaram ganhos de desempenho de 30% para tuple-at-a-time, 11% em column-at-a-time e 6× na execução vector-at-a-time quando comparados com o x86.*

1. Introdução

Nas últimas décadas, a disparidade entre o desempenho do processador e a latência da memória principal cresceu fortemente, um problema bem conhecido chamado *"memory wall"* [Wulf and McKee 1995]. Esta lacuna crescente apresenta impacto direto no processamento de dados em larga escala, especialmente em bancos de dados em memória. Ao longo dos anos, os bancos de dados em memória, tornaram-se populares devido a queda no custo da DRAM e ao crescimento de sua capacidade de megabytes para terabytes. No entanto, quando aplicações com comportamento de *streaming* movem grande quantidade de dados através da hierarquia de memória, elas sofrem penalidades pela latência das interconexões e da cache.

Para atenuar o problema de movimentação de dados, a abordagem de processamento em memória (PIM - Processing-in-Memory) [Kautz 1969] inverte o fluxo de processamento de dados, movendo as instruções para onde os dados residem. Recentemente, o lançamento do Cubo de Memória Híbrida (HMC - Hybrid Memory Cube) tornou o PIM tangível para aplicações orientada a dados [Balasubramonian et al. 2014]. O HMC utiliza tecnologia de integração 3D, unindo 4 ou 8 camadas de DRAM a uma camada de lógica usando interconexões através do silício (TSVs - Through-Silicon Vias) [Beyne et al. 2008]. Essa memória é dividida em 32 partições independentes chamadas de *vaults*. A camada lógica controla os bancos DRAM e também permite a execução de instruções com alto nível de paralelismo entre os *vaults*.

O HMC pode ser usado como uma memória principal simples (substituindo as DDR 3), proporcionando em média 10$\times$ melhor desempenho e 70% menos consumo de energia. Além disso, também apresenta um conjunto de instruções de atualização formadas por operações de leitura-operação ou leitura-modificação-escrita sobre dados de 8 ou 16 bytes [Jeddeloh and Keeth 2012]. No entanto, o atual conjunto de instruções não é otimizado para operar sobre grandes volumes de dados, como os presentes em bancos de dados [Santos et al. 2017]. As memórias DRAM presentes no HMC utilizam linhas (também chamadas de páginas) de 256 bytes com política de página fechada, a qual fecha a linha após a mesma receber o último acesso, ou seja, não houver nenhum outro acesso à uma mesma linha dentro dos buffers de requisição. Isso significa que o HMC favorece acessos aleatórios (linhas diferentes). No entanto, os acessos posteriores para a mesma linha sofrerão alta latência para reabrir a linha fechada recentemente. Tais acessos tardios geralmente acontecem ao executar várias instruções de 16 bytes de dados por vez. Tais instruções podem chegar tardiamente devido a contenções nos buffers e interconexões existentes entre o processador e a memória principal.

Neste trabalho, é investigado o conjunto de instruções do HMC para executar as operações de seleção em banco de dados com uma abordagem de processamento próximo a memória [Khoram et al. 2017]. Ademais, são apresentadas as seguintes contribuições:

1. Extensão do conjunto de instruções do HMC para aproveitar ao máximo a arquitetura DRAM interna do HMC ao processar operações de seleção.
2. Análise de desempenho de execução das operações de seleção dentro do HMC.
3. Avaliação os prós e contras das extensões propostas ao executar os principais mecanismos de consulta: *tuple-at-a-time, column-at-a-time, vector-at-a-time*.

O restante deste artigo está organizado da seguinte forma: A Seção 2 apresenta os

trabalhos relacionados. A Seção 3 apresenta a proposta. A seção 4 descreve a metodologia e os resultados parciais. A seção 5 descreve as próximas etapas deste trabalho.

2. Trabalhos Relacionados

O problema de *"memory wall"* motivou diversos trabalhos nas últimas décadas com foco em algoritmos para melhor explorar os benefícios da cache [Wulf and McKee 1995, Boncz et al. 1999].

No contexto do processamento de dados próximo a memória, [Xi et al. 2015] apresenta o acelerador para SGBD externo a DRAM chamado JAFAR, o qual envia as operações de seleção de 64 bits para processamento em memórias DDR 3. Já com a utilização do HMC, o processador envia as instruções para a camada lógica sem necessidade de hardware externo, além de operar sobre vetores de até 256 bytes por instrução.

[Mirzadeh et al. 2015] apresenta outra abordagem para colocar um acelerador dentro da camada lógica do HMC, porém com o objetivo de suportar algoritmos de junção. Este trabalho remodela os algoritmos de junção de *hash* e de *merge* para minimizar o acesso de uma única palavra (por exemplo, 16 bytes) evitando o re-acesso da linha de DRAM. Por outro lado, não considera as modificações necessárias no HMC para executar tais operações para SGBDs do tipo *row-store*.

Em outros trabalhos recentes, foi considerado o uso de grandes unidades funcionais vetoriais com bancos de registradores dentro do HMC [Alves et al. 2016], estáticas ou com capacidade de reconfiguração [Santos et al. 2017]. No entanto, esse design requer um controle fino do programador para escolher o melhor tamanho do operando. Estes trabalhos consistem em uma modelagem extremamente cara em termos de hardware, porque requerem muita lógica extra para suportar processamento de vetores de 8 KB.

Diante disso, na presente dissertação propõe-se uma abordagem que estende as instruções do HMC para operar sobre dados maiores reduzindo as operações de abertura de linha DRAM e melhor explorando a camada lógica disponível.

3. Processamento de predicados no HMC

Nesta seção, é discutida a extensão proposta para a camada lógica do HMC com o objetivo de executar as operações de seleção. Primeiro, é feita uma avaliação em torno da migração do processamento da seleção para HMC e em seguida é apresentada a modificação arquitetural proposta. Para essa investigação foi implementada a consulta 06 do TPC-H em linguagem C e depois traduzida para o assembly a ser executado, como mostrado na figura 1. A consulta 06 executa a seleção em três atributos da tabela *lineitem*.

O processamento de predicados consiste em operações de comparação de registros como mostrado na Figura 1 representando a primeira comparação do atributo *l_shipdate* na consulta 06. Neste caso, é apresentado uma remodelagem no fluxo de processo da seleção, substituindo a instrução de comparação (*CMP*) x86 pela instrução HMC recíproca (*HMC_CMP*), a qual envia os bits de status da operação de volta para o processador x86. Nesta abordagem de PIM, o processador continua a buscar e desencadear as instruções, mas a execução e o acesso aos dados dependem da camada lógica do HMC, sem sofrer da latência imposta pela hierarquia de cache.

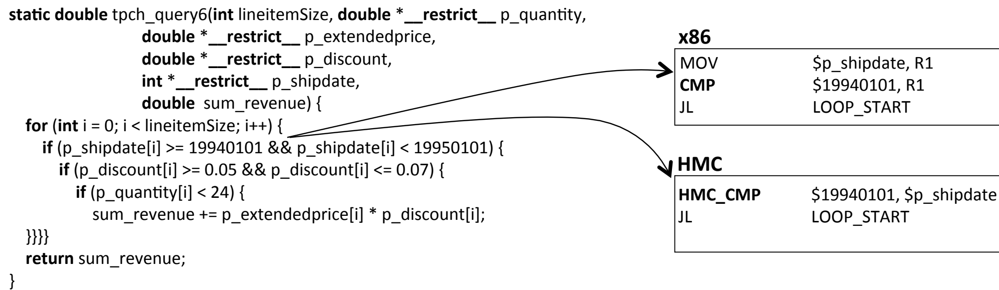
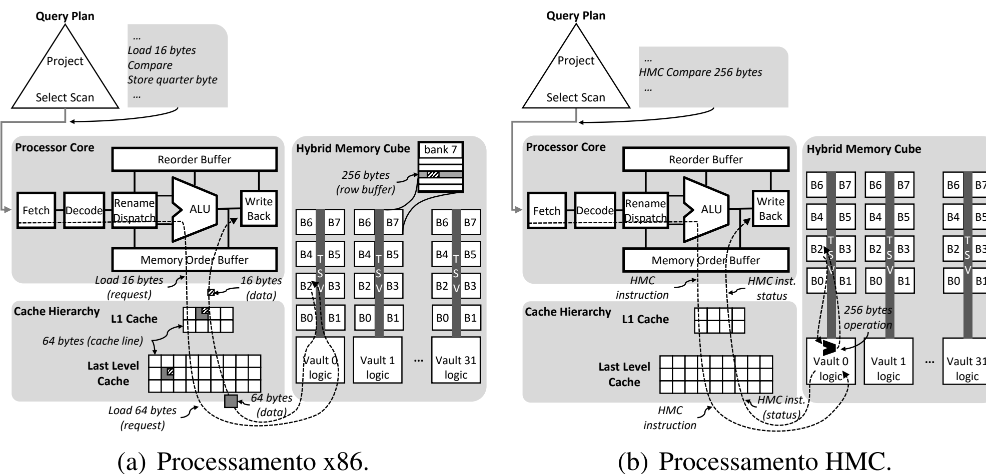


Figura 1. Implementação em linguagem C da consulta 06 do TPC-H e trecho de código assembly correspondente ao primeiro predicado.

A Figura 2(a) ilustra o processamento de predicados em uma operação de seleção com as instruções da arquitetura x86 atuais e utilizando o HMC como memória principal (reiterando que o HMC trabalha como uma pilha de DRAMs). Neste cenário de processamento, as operações subjacentes do SGBD continuam sem modificação, com o mesmo modelo de processamento de consulta. A operação de seleção envia um predicado para qualificar os dados. Dessa forma, as instruções subjacentes do x86 são alocadas no *pipeline* do processador. Na arquitetura x86 atual com instruções AVX-128, cada operação irá trabalhar sobre 16 bytes de dados. No primeiro acesso a cada linha de cache, uma falta de dados nas caches L1, L2 e L3 irá requerer um acesso à memória, já que a hierarquia de cache sempre trabalha com linhas de cache (64 bytes). Após a memória retornar a linha de cache, o processador poderá operar sobre 16 bytes a cada instrução.



(a) Processamento x86.　　(b) Processamento HMC.

Figura 2. Processamento de predicados com 86 vs. HMC

A Figura 2(b) descreve a proposta de seleção para operar em toda a linha da DRAM de 256 bytes. Nessa abordagem, quando uma instrução HMC é identificada no pipeline do processador, essa instrução é enviada para ser executada no HMC (sem sofrer latência da hierarquia de cache). A camada lógica solicita até 256 bytes por requisição ao banco DRAM para executar cada instrução. Uma vez que a instrução é processada, um status é enviado para o processador. A operação de seleção termina quando o processador armazena a saída usando instruções x86 seguindo o plano de consulta tradicional.

4. Experimentos

Nesta seção são apresentados o ambiente de simulação, a metodologia experimental e os resultados com a implementação da operação de seleção sobre os mecanismos de consulta *tuple/column/vector-at-a-time*.

4.1. Metodologia e Configuração

Para avaliar a proposta foi utilizado um simulador com precisão de ciclos chamado SiNUCA [Alves et al. 2015]. O SiNUCA permite modelar o nosso tamanho de operação personalizado até 256 bytes dentro do HMC, para assim, entendermos o comportamento arquitetural ao executar a operação de seleção.

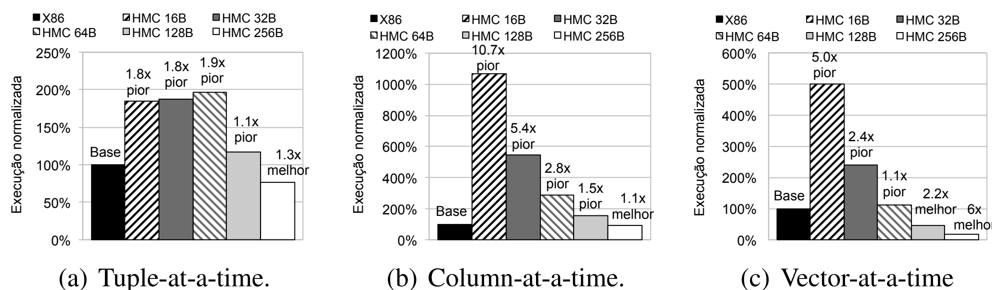
A arquitetura do cenário base é inspirada na micro-arquitetura do processador Intel Sandy Bridge referente ao x86. O Sandy Bridge foi configurado com 16 cores com conjunto de instruções AVX-128 modelando parâmetros iguais aos utilizados pelo autor na validação do simulador [Alves et al. 2015]. Os parâmetros para memória principal utilizada foram baseados na versão 2.0 do HMC [HMC-Consortium 2017]. Nos experimentos do presente trabalho, foi gerado um banco de dados TPC-H de 1 GB, além disso, foi selecionado um micro-benchmark executando a consulta 06 do TPC-H. A consulta 06 implementa expressões booleanas complexas, o que resulta na possibilidade de empurrar para baixo os predicados mais seletivos no plano de consulta.

A abordagem proposta neste trabalho utiliza o conjunto atual de instruções do HMC estendendo apenas, o tamanho dos operadores de 16 bytes para tamanhos de até 256 bytes. Neste caso, estamos executando as instruções *load* e *compare* dentro do HMC entrelaçado com as instruções x86 tradicionais.

4.2. Resultados Preliminares

Os resultados de desempenho para o cenário base x86 e HMC executando a consulta 06 são apresentados na figura 3. O HMC foi avaliado utilizando cinco tamanhos de instrução diferentes, 16, 32, 64, 128 e 256 bytes, enquanto o x86 foi configurado para 16 bytes (ou seja, o maior tamanho de instrução x86 suportado). As instruções de *store* são executadas com assistência da cache nos dois casos (x86 e HMC), enquanto as instruções *load-compare* são feitas na camada lógica do HMC quando não utilizamos o x86 puro.

Tuple-at-a-time: A figura 3(a) apresenta os resultados para a execução *tuple-at-a-time* no modelo de armazenamento *row-store*. Ao executar operações de 16 bytes de



(a) Tuple-at-a-time. (b) Column-at-a-time. (c) Vector-at-a-time

Figura 3. Tempo de execução normalizado para a operação de seleção da consulta 06 do TPC-H

largura no HMC, o tempo para processar a seleção aumentou em 84% em relação ao x86. Observou-se o mesmo resultado ao executar com operações de tamanho 32 e 64 bytes. Estas operações aumentaram o tempo de resposta em 88% e 97%, respectivamente, em comparação com o x86. Como esperado, o aumento do tempo de resposta foi devido à quantidade de acessos à DRAM para abrir e fechar a linha de dados ou seja, a política de página fechada. Além disso, observa-se que cada instrução de *load* x86 obtém uma linha de cache (64 Bytes) da DRAM e a relação entre o tamanho da tupla e o tamanho da operação faz com que cada operação processe apenas uma tupla de 64 bytes. Considerando a operação HMC de 128 bytes, o tempo de execução aumenta apenas em 16%, pois duas tuplas de 64 bytes são avaliadas por operação. O melhor cenário ocorre quando o tamanho da instrução está configurado para 256 bytes. O tempo de execução caiu em 30% em comparação com cenário base x86, já que a seleção pode processar 4 tuplas contíguas por operação sem sofrer a latência da hierarquia de cache.

Column-at-a-time: A figura 3(b) apresenta os resultados para a execução *column-at-a-time* no modelo de armazenamento *column-store*. A execução *column-at-a-time* é mais eficiente em uso de CPU do que a *tuple-at-a-time*, porque não há desperdício em *load* de colunas não utilizadas, ou seja, dados de uma mesma coluna agora estão contíguos na memória. Além disso, apenas o primeiro predicado processa todos os valores de uma determinada coluna. Os predicados remanescentes apenas processam as correspondências da coluna anterior evitando muitos acessos a DRAM.

O tempo de execução em comparação com x86 aumentou em 10,7×, 5,44×, 2,84× e 1,55× ao processar no HMC com operações de 16, 32, 64 e 128 bytes respectivamente, dado o número de re-acessos a mesma linha na DRAM. O HMC demonstrou melhorias de desempenho de 11% em relação ao cenário base x86 com o tamanho da operação em 256 bytes para tirar proveito do tamanho da linha de DRAM disponível e mitigar o impacto da política de página fechada. Observa-se que após o processamento da primeira coluna, o processador precisa buscar a máscara de bits gerada anteriormente para decidir as porções da segunda coluna que precisam ser processadas. Gerando assim, uma dependência de dados e atrasando o envio de novas instruções para o HMC.

Vector-at-a-time: A figura 3(c) apresenta os resultados para a estratégia *vector-at-a-time* no modelo de armazenamento *column-store*. A estratégia *Vector-at-a-time* é a mais eficiente em uso de CPU e com o melhor uso de cache segundo a literatura. A avaliação com o HMC de 16, 32 e 64 bytes aumentou o tempo de execução em 5×, 2,41× e 1,11×, respectivamente, em comparação com o x86. Na execução x86 atual, a cache é preenchida com muitos dados que são pouco utilizados durante o processamento de diferentes predicados. Mesmo com os esforços para melhorar a localidade espacial de cache com o uso da estratégia *vector-at-a-time*, a localidade temporal baixa para as diferentes colunas faz com que a latência devido as faltas em cache cause aumento no tempo de execução. As execuções usando operações de 128 e 256 bytes superaram o cenário base x86 em 2,19× e 6,00× respectivamente. Diferentemente da execução *column-at-a-time*, a *vector-at-a-time* avalia os predicados executando apenas *loads* que filtram os valores. Tal modelo, armazena o resultado do *bitmap* somente após o processamento do último predicado. Uma vez que a presente proposta substitui apenas as instruções de *load-compare*, menos instruções x86 são intercaladas com o HMC, permitindo que o HMC processe todos os valores de cada vetor de coluna e apenas armazene o resultado final.

5. Trabalhos Futuros

O cronograma para finalizar essa dissertação inclui uma modelagem de registradores na camada lógica do HMC para oferecer operações de *load*, *operation* e *store* em endereços distintos de memória, uma vez que atualmente apenas operações de atualização sobre o mesmo endereço são suportadas. Além disso, pretendemos incluir operações de predicação transferindo parte do controle de execução para o HMC. Por fim, pretendemos apresentar resultados de execução com outras consultas do TPC-H incluindo variações da seletividade dessas consultas.

Agradecimentos

Este trabalho foi parcialmente financiado pelo CAPES e CNPq, concedido por 441944/2016-0.

Referências

Alves, M. A. Z., Diener, M., Santos, P. C., and Carro, L. (2016). Large vector extensions inside the hmc. In *DATE*, pages 1249–1254.

Alves, M. A. Z., Villavieja, C., Diener, M., and t al. (2015). Sinuca: A validated micro-architecture simulator. *HPCC*, pages 605–610.

Balasubramonian, R., Chang, J., Manning, T., and et al. (2014). Near-data processing: Insights from a MICRO-46 workshop. *IEEE Micro*, pages 36–42.

Beyne, E., Moor, P. D., Ruythooren, W., and et al. (2008). Through-silicon via and die stacking technologies for microsystems-integration. *IEDM*, page 1–4.

Boncz, P. A., Manegold, S., and Kersten, M. L. (1999). Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65.

HMC-Consortium (2017). Hmc specification 2.1.

Jeddeloh, J. and Keeth, B. (2012). Hybrid memory cube new dram architecture increases density and performance. In *(VLSI)*, pages 87–88.

Kautz, W. H. (1969). Cellular logic-in-memory arrays. *IEEE Trans. Comput.*, pages 719–727.

Khoram, S., Zha, Y., Zhang, J., and Li, J. (2017). Challenges and opportunities: From near-memory computing to in-memory computing. In *ISPD*, pages 43–46.

Mirzadeh, N. S., Kocberber, O., Falsafi, B., and Grot, B. (2015). Sort vs. hash join revisited for near-memory execution. In *ASBD*.

Santos, P. C., Oliveira, G. F., Tome, D. G., and et al. (2017). Operand size reconfiguration for big data processing in memory. In *DATE*, pages 710–715.

Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. *SIGARCH*, 23(1):20–24.

Xi, S. L., Babarinsa, O., Athanassoulis, M., and Idreos, S. (2015). Beyond the wall: Near-data processing for databases. In *DAMON*, pages 2:1–2:10.