Execução Energeticamente Eficiente de Aplicações Estêncil com o Processador *Manycore* MPPA-256

Emmanuel Podestá Jr.¹, Alyson D. Pereira¹, Rodrigo C. O. Rocha², Márcio Castro¹, Luís F. W. Góes²

¹ Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD) Universidade Federal de Santa Catarina (UFSC) – SC, Brasil

² Grupo de Computação Criativa e Paralela (CreaPar) Pontifícia Universidade Católica de Minas Gerais (PUC Minas) – MG, Brasil

emmanuel.podesta@grad.ufsc.br, alyson.pereira@posgrad.ufsc.br, rcor@pucminas.br, marcio.castro@ufsc.br, lfwgoes@pucminas.br

Resumo. Neste artigo é proposta uma adaptação do framework PSkel para o processador manycore de baixa potência MPPA-256. O framework permite simplificar o desenvolvimento de aplicações estêncil iterativas para o MPPA-256, escondendo do desenvolvedor detalhes de implementação. Os resultados obtidos no MPPA-256 mostraram uma redução do consumo de energia de aplicações estêncil iterativas de até 1.45x em comparação com um processador multicore Intel Broadwell.

1. Introdução

Plataformas de Computação de Alto Desempenho (CAD) tem sido avaliadas quase que exclusivamente pela suas capacidades de processamento. Contudo, o consumo excessivo de energia é uma barreira para o aumento de desempenho de forma escalável nestas plataformas. Por essa razão, o estudo de técnicas que melhorem a eficiência energética em plataformas de CAD está se tornando muito importante. Recentemente, uma nova classe de processadores *manycore* de baixa potência tais como o Sunway SW26010 [Fu et al. 2016] e o Kalray MPPA-256 [Castro et al. 2013] foram desenvolvidos. Esses processadores possuem centenas de núcleos de processamento capazes de lidar com paralelismo de dados e tarefas com baixo consumo de energia.

Processadores *manycore* de baixa potência (*low-power manycore processors*) apresentam uma melhor eficiência energética em comparação com processadores de propósito geral presentes atualmente [Francesquini et al. 2014], contudo as suas características arquiteturais tornam o desenvolvimento de aplicações uma tarefa desafiadora [Varghese et al. 2014, Castro et al. 2016, Castro et al. 2014]. Geralmente, núcleos de processamento sem coerência de *cache* são distribuídos em uma arquitetura organizada em *clusters*, onde cada *cluster* possui uma memória local (compartilhada somente entre os núcleos do *cluster*). Dessa forma, a comunicação entre *clusters* deve que ser efetuada através de uma *Network-on-Chip* (NoC) de maneira distribuída. Por essa razão, o tempo de comunicação pode variar entre os núcleos que estão se comunicando.

Uma possível abordagem para facilitar o desenvolvimento de aplicações paralelas para processadores *manycore* de baixa potência é através do uso de abstrações de mais alto nível fornecidas por padrões paralelos ou esqueletos algorítmicos [Cole 2004]. Esses padrões permitem que desenvolvedores foquem na construção de algoritmos, sem a preocupação com problemas de sincronização ou escalonamento de tarefas. Esses problemas são resolvidos de forma transparente pelo *framework* do padrão adotado.

Dentre os diversos padrões paralelos existentes (e.g., *map*, *reduce*, *pipeline* e *scan*), o padrão estêncil tem sido muito utilizado em várias áreas importantes, como física quântica, previsão do tempo e processamento de imagens [Gonzalez and Woods 2006, Holewinski et al. 2012, Lutz et al. 2013]. No padrão estêncil, para cada elemento de uma estrutura *n*-dimensional de entrada é computado um novo valor para o respectivo elemento em uma estrutura *n*-dimensional de saída, utilizando-se como base os valores dos elementos vizinhos ao elemento de entrada. A quantidade de vizinhos e a computação a ser realizada em cada elemento é definida por uma função (ou *kernel*) estêncil. Em aplicações estêncil iterativas, os valores produzidos na estrutura *n*-dimensional de saída em uma iteração *i* são utilizados como entrada da iteração *i* + 1.

Alguns *frameworks* foram propostos para o desenvolvimento de aplicações paralelas com base no padrão estêncil, tais como SkelCL [Steuwer et al. 2011], SkePU [Enmyren and Kessler 2010] e PSkel [Pereira et al. 2015]. Em especial, o *framework* PSkel provê uma abstração de alto nível para o desenvolvimento de aplicações estêncil em ambientes heterogêneos compostos por processadores *multicore* e *Graphical Processing Units* (GPUs). Todavia, nenhum desses *frameworks* possui suporte para processadores *manycore* de baixo potência de energia emergentes tais como o MPPA-256.

Portanto, nesse artigo é proposta uma adaptação completa do *framework* PSkel para o processador MPPA-256, a qual permite simplificar significativamente o desenvolvimento de aplicações estêncil nesse processador. A adaptação permite eliminar as dificuldades de desenvolvimento intrínsecas do processador, fazendo com que as aplicações já implementadas em PSkel possam ser executadas no MPPA-256 sem a necessidade de nenhuma modificação em seus códigos. Os resultados obtidos mostram que o MPPA-256 apresenta uma melhor eficiência energética que um processador Intel Broadwell com 10 núcleos físicos ao executar três aplicações estêncil implementadas no PSkel.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta os principais conceitos do processador *manycore* MPPA-256 e do *framework* PSkel. A Seção 3 discute a adaptação do *framework* PSkel para fornecer suporte ao MPPA-256. Os resultados obtidos com a adaptação do *framework* PSkel para o MPPA-256 são apresentados na Seção 4. Por fim, a Seção 5 apresenta os trabalhos relacionados ao tema abordado por esse artigo e Seção 6 apresenta as conclusões deste trabalho.

2. Fundamentação Teórica

2.1. MPPA-256

O MPPA-256 é um processador *manycore* desenvolvido pela empresa francesa Kalray, o qual possui 256 núcleos de processamento de 400 MHz denominados *Processing Elements* (PEs). Além dos PEs, o processador possui 32 núcleos dedicados a gerência de recursos denominados *Resource Managers* (RMs). PEs e RMs são distribuídos fisicamente no *chip* em 16 *clusters* e 4 subsistemas de Entrada/Saída (E/S), contendo cada *cluster* 16 PEs e 1 RM. Além dos *clusters*, o MPPA-256 possui 4 subsistemas de E/S contendo, cada um, 4 RMs. Toda a comunicação entre *clusters* e/ou subsistemas de E/S é feita através de uma NoC *torus* 2D. A arquitetura do MPPA-256 pode ser vista na Figura 1a.

A finalidade principal dos PEs é executar *threads* de usuário de forma ininterrupta e não preemptível para realização de computação. PEs de um mesmo *cluster* compartilham uma memória de 2 MB, a qual é utilizada para armazenar os dados a serem processados pelos PEs. Cada PE possui também uma memória *cache* associativa 2-*way* de 32KB para dados e uma para instruções. Porém, o processador não dispõe de coerência



Figura 1. Visão geral do MPPA-256 (esquerda) e uma ilustração do padrão estêncil oferecido pelo PSkel (direita).

de *caches*, o que dificulta o desenvolvimento de aplicações para esse processador. Por outro lado, a finalidade dos RMs é gerenciar E/S, controlar comunicações entre *clusters* e/ou subsistemas de E/S e realizar comunicação com uma memória RAM. Na arquitetura utilizada neste artigo, um dos subsistemas de E/S está conectado a uma memória externa *Low Power Double Data Rate 3* (LPDDR3) de 2 GB.

Trabalhos anteriores mostraram que desenvolver aplicações paralelas otimizadas para o MPPA-256 é um grande desafio [Francesquini et al. 2014] devido a alguns fatores importantes. O primeiro deles está relacionado ao modelo de programação híbrido exigido pelo processador: threads em um mesmo cluster se comunicam através de uma memória compartilhada local, porém a comunicação entre *clusters* é feita explicitamente via NoC, em um modelo de memória distribuída. Mais especificamente, aplicações desenvolvidas para o MPPA-256 precisam utilizar duas bibliotecas de programação paralela para utilizar os recursos do processador: OpenMP, baseado em um modelo de memória compartilhada, utilizada para paralelizar a computação dentro de cada *cluster* e uma Application Programming Interface (API) proprietária, que segue um modelo de memória distribuída, sendo utilizado na comunicação entre os *clusters* e o subsistema de E/S por meio da NoC. O segundo fator importante está relacionado a capacidade limitada de memória no chip: cada cluster possui apenas 2 MB de memória local de baixa latência. Portanto, aplicações reais precisam constantemente realizar comunicações com o subsistema de Entrada e Saída (E/S) conectado à memória LPDDR3. Por fim, o último fator está diretamente relacionado à ausência de coerência de cache: cada PE possui uma memória *cache* privada sem coerência de *cache*, sendo necessário o uso explícito de instruções do tipo *flush* para atualizar a *cache* de um PE quando necessário.

2.2. PSkel

O PSkel é um *framework* de programação em alto nível para o padrão estêncil, baseado no conceito de esqueletos paralelos. Ele oferece suporte à execuções paralelas em arquiteturas heterogêneas incluindo CPU e GPU. Utilizando uma única interface de programação escrita em C++, o usuário é responsável por definir o *kernel* principal da computação estêncil, enquanto o *framework* se encarrega de gerar código executável para as diferentes plataformas paralelas, realizando todo o gerenciamento de memória e transferência de dados entre dispositivos de forma transparente [Pereira et al. 2015].

A API do PSkel possibilita a definição de *templates* para a manipulação de estruturas *n*-dimensionais, denominadas Array (1 dimensão), Array2D (2 dimensões) e Array3D (3 dimensões). Além disso, o *framework* provê abstrações para a definição da vizinhança do estêncil (Mask) e o *kernel* da computação estêncil (stencilKernel()). O stencilKernel() é um método a ser implementado pelo usuário que descreve, especificamente, a computação que será executada para cada célula do Array de entrada com base nos valores de sua vizinhança (Mask).

Em uma aplicação estêncil iterativa, cada iteração utiliza a máscara de vizinhança (Mask) sobre o Array de entrada para determinar o valor de cada célula do Array de saída. No exemplo da Figura 1b, o valor de cada célula do Array de saída é determinado em função dos valores de cada uma das células vizinhas adjacentes. Esse processo é realizado para todas as células do Array de entrada, produzindo um Array de saída da computação estêncil. Ao final de uma iteração, o Array de saída será considerado como Array de entrada para a próxima iteração no caso de uma aplicação estêncil iterativa.

3. Adaptação do framework PSkel para o MPPA-256

A adaptação do *framework* PSkel para o processador MPPA-256 proposta neste artigo segue um modelo mestre/escravo. Um processo mestre é executado no subsistema de E/S conectado à memória LPDDR3 de 2 GB, sendo responsável por alocar o Array de entrada e por distribuir os dados entre os processos escravos. Em cada *cluster* é instanciado um único processo escravo que é responsável por gerenciar a computação no seu *cluster*. Devido às limitações de memória dos *clusters* (apenas 2 MB por *cluster*), o processo mestre deve subdividir o Array de entrada em blocos denominados *tiles* e, então, gerenciar as comunicações dos mesmos com os processos escravos.

O processo mestre particiona o Array de entrada com dimensão n em b blocos, onde b é o número de *clusters* utilizados na computação. Então, cada bloco é particionado em *tiles* de tamanho fixo definidos pelo usuário. Quando são feitas computações estêncil sobre o *tile*, dependências de vizinhança, inerentes ao padrão paralelo do estêncil, precisam ser consideradas durante o particionamento dos dados. Uma das principais soluções para satisfazer essas dependências é via blocos sobrepostos, resultando em dados redundantes e computação por *tile* [Meng and Skadron 2011, Holewinski et al. 2012, Rocha et al. 2017]. Essa técnica é muito importante em *manycores* de baixa potência como o MPPA-256, onde o sobrecusto de comunicação pode ser elevado. O impacto dos custos de comunicação será analisado posteriormente na Seção 4.

Portanto, foi implementada uma técnica de *tiling* trapezoidal. Para ilustrar e detalhar como essa técnica de *tiling* pode ser aplicada na computação estêncil, utilizamos a definição a seguir. Seja A um Array2D, com dimensões dim(A) = (w, h), onde w e h são, respectivamente, a largura e a altura. Utilizando *tiles* de dimensões (w', h') produz $\lceil \frac{w}{w'} \rceil \lceil \frac{h}{h'} \rceil$ *tiles* possíveis de A. Seja $A_{i,j}$ um *tile*, onde $0 \le i < \lceil \frac{w}{w'} \rceil$ e $0 \le j < \lceil \frac{h}{h'} \rceil$. $A_{i,j}$ possui offset (iw', jh') relativo ao canto superior esquerdo de A e dim $(A_{i,j}) = (\min\{w', w - iw'\}, \min\{h', h - jh'\})$. O offset é uma indexação de deslocamento necessário para acessar os elementos do *tile* (Figura 2). Essa técnica pode ser facilmente estendida para mais dimensões.

Aplicar um estêncil em A envolve aplicar a função de vizinhança (máscara) contendo o deslocamento de cada vizinho de um dado elemento central. Por causa da dependência entre vizinhos, para computar a função estêncil, de acordo com as limitações necessárias pelos *tiles*, se torna necessário obter valores de *tiles* adjacentes. Seja r o *range* da máscara de vizinhos, i.e., r é o deslocamento mais distante necessário para a vizinhança definida pela máscara. A área de r envolvendo a vizinhança é denominada região *halo*. Se a função estêncil é aplicada iterativamente sobre A, para t iterações,



Figura 2. Diagrama do *tiling* 2D. Um *tile* lógico (linha interna sólida) é contido dentro do Array 2D (linha externa pontilhada) com *offsets* verticais e horizontais dado por jh' e iw'. Computar t' consecutivas iterações estêncil no *tile* requer um aumento no *tile* lógico com uma *ghost zone* (área entre a linha interna sólida e a linha externa sólida), que é constituída de regiões *halo* (área entre a linha interna sólida).



Figura 3. Exemplo do funcionamento do método strides do MPPA-256.

a dependência da vizinhança entre os *tiles* limita o número de iterações que podem ser computadas consecutivamente sem a necessidade de realizar comunicações entre *tiles*.

Além disso, devido à restrição da API e NoC no MPPA-256, os dados armazenados em cada *tile* precisam ser contíguos para serem transferidos pela NoC. A fim de se evitar cópias locais de dados, o que desperdiçaria memória e tempo de processamento, utiliza-se o conceito de comunicação por *strides*. Cada *stride* é uma parte contígua do Array original, sendo determinado por deslocamentos (*offsets*) especificados durante a execução. Então, cada *stride* é enviado para o Array de entrada em um processo escravo na posição determinada de acordo com o tamanho dos *tiles* e do Array original. O método de *strides* possibilita a definição de algumas variáveis para o gerenciamento do Array, sendo a mais importante os *offsets* que serão utilizados. A partir deles, o método irá efetuar a comunicação de maneira direta para oArray destino. Como dito anteriormente, a utilização de *tiles* aumentados permite reduzir a quantidade de comunicações necessárias entre os processos mestre e escravos. Fazendo o aumento dos *tiles* em uma dimensão temporal, os processos escravos podem executar múltiplas iterações sem a necessidade de comunicação ou sincronização com o processo mestre. O escalonamento dos *tiles* nos *clusters* é feito de maneira circular (*round-robin*). Devido a isso, alguns *clusters* podem receber mais *tiles* que outros, dependendo do número de *tiles* e *clusters* usados na computação. Toda a comunicação entre o mestre e os escravos é feita utilizando-se a API de comunicação assíncrona oferecida pelo processador MPPA-256. A comunicação com cada processo escravo é feita de forma individual, mapeando diretamente áreas contíguas de memória de seus respectivos *tiles*. Além disso, a implementação atual permite a execução de aplicações estêncil iterativas. Nesse caso, o escalonamento dos *tiles* e a computação dos mesmos pelos *clusters* é repetida a cada iteração da aplicação.

Cada processo escravo realiza a computação do *tile* recebido no *cluster* utilizando o *kernel* de computação estêncil definido pelo usuário. A paralelização da computação dentro do *cluster* é feita com auxílio da API OpenMP. Em cada *cluster* podem ser criadas até 16 *threads* (uma para cada PE), onde cada uma é responsável por executar o *kernel* estêncil em um subconjunto de elementos dos *tiles*. Quando a computação do *kernel* estêncil é finalizada, os *tiles* resultantes são enviados pelos processos escravos para o processo mestre, onde são agrupados em um único Array, constituindo o resultado final da computação estêncil em uma iteração. Tendo em vista que os *tiles* são regiões contíguas na memória, processos escravos precisam gerenciar os *offsets* de dados para escrevê-los nas posições corretas no Array no processo mestre. Esse gerenciamento é efetuado pela API do MPPA-256, mais especificamente, pelo método de *strides* mencionado anteriormente. A Figura 3 ilustra esse procedimento para o caso de um Array2D.

Para fornecer uma maior facilidade ao usuário, todas as tarefas complexas relacionadas com a técnica de *tiling*, comunicações NoC e adaptações discutidas nessa seção são abstraídas, pois elas são incluídas no *back-end* do PSkel. Isso significa que aplicações desenvolvidas com o *framework* PSkel podem executar no MPPA-256 sem a necessidade de modificações no código fonte.

4. Resultados Experimentais

Nesta seção será avaliado o desempenho e o consumo energético de aplicações estêncil do PSkel quando executadas no MPPA-256 e em um processador Intel Xeon E5-2640 v4 com 10 núcleos de 2.4GHz (Broadwell). As medições de energia no MPPA-256 foram feitas considerando todos os *clusters*, a memória, os subsistemas de E/S e a NoC, onde foram coletadas por meio dos sensores de potência e energia disponíveis no MPPA-256. Como o MPPA-256 possui características intrínsecas do próprio processador que garante baixa variabilidade entre as execuções, foram realizadas somente 5 repetições de cada experimento, computando-se a média aritmética dos valores. Todos os experimentos no MPPA-256 consideraram 16 PEs por *cluster*. Por outro lado, as medições de consumo de energia foram feitas no processador Intel com uso do *Running Average Power Limit* (RAPL) através da biblioteca PAPI [Weaver et al. 2012]. Em cada experimento foram utilizadas 10 *threads* (uma *thread* por núcleo) sem uso de *hyperthreading*. Cada experimento foi repetido 30 vezes e a média aritmética dos resultados foi calculada. Todos os resultados (MPPA-256 e Intel) apresentaram um desvio-padrão menor que 1%.

É possível reduzir a quantidade de sincronizações e comunicações realizadas na solução para o MPPA-256 de duas maneiras: aumentando o tamanho dos *tiles* ou aumentando a quantidade de iterações sobre cada *tile*. Como podemos ver na Figura 2, ao aumentarmos a quantidade de iterações sobre o *tile*, precisamos aumentar o *tile* lógico, formando um *tile* aumentado que será enviado para o *cluster*. Desta forma, devido às limitações de memória em cada *cluster*, ao ser especificado uma quantidade muito

grande de iterações, o *tile* aumentado enviado para o escravo pode ser maior que o limite de memória de cada *cluster*. Portanto, foi fixado uma quantidade de 10 iterações sobre cada *tile*. Além disso, para os experimentos terem uma quantidade significativa de sincronizações sobre aplicações estêncil iterativas, foi adotado 30 iterações para cada aplicação.

4.1. Aplicações Estêncil

Para a realização dos experimentos foram utilizadas as seguintes aplicações estêncil:

Fur: modela a formação de padrões sobre a pele de animais¹. Nessa aplicação, a pele do animal é modelada por uma *array* bidimensional de células de pigmento que podem estar em um dos dois estados: colorida ou não-colorida. As células coloridas secretam ativadores e inibidores. Ativadores fazem uma célula central se tornar colorida; inibidores, por outro lado, fazem uma célula central se tornar não colorida. A diferença entre as potências dos ativadores e inibidores resulta em uma célula colorida e mais inibidores resulta em uma célula colorida e mais inibidores resulta em uma célula não colorida. Nos casos em que as potências dos ativadores e inibidores forem iguais, a cor da célula permanece inalterada. A máscara contém células adjacentes à célula central e seu tamanho é parametrizável. Neste trabalho foi utilizado 2 vizinhos adjacentes em cada direção.

Jacobi: método iterativo para resolver equações matriciais [Demmel 1997]. O método converge garantidamente se a matriz de entrada é restrita ou irredutívelmente dominante diagonalmente, i.e., $|u_{i,i}| > \sum_{j \neq i} |u_{i,j}|$, para todo *i*. A Equação 1 define a computação em cada passo do método iterativo de Jacobi para resolver a equação discreta elíptica de Poisson [Demmel 1997]. A solução aproximada é computada discretizando o problema na matriz em pontos espaçados de forma equivalente por $n \times n$.

$$u_{i,j}' = \frac{u_{i\pm 1,j} + u_{i,j\pm 1} + h^2 f_{i,j}}{4} \tag{1}$$

A cada passo, o novo valor de $u_{i,j}$ é obtida fazendo a média $h^2 f_{i,j}$ dos seus vizinhos, onde $h = \frac{1}{n+1}$ e $f_{i,j} = f(ih, jh)$, para uma dada função f.

GoL: autômato celular que implementa o Jogo da Vida de Conway [Gardner 1970]. O autômato é representado por um *array* bidimensional, onde cada elemento representa um indivíduo vivo ou um indivíduo morto. A máscara do estêncil, a qual determina a interação entre o indivíduo e seus vizinhos, considera as 8 células vizinhas adjacentes à célula central. Dependendo dos valores dos vizinhos, o elemento pode modificar seu estado entre vivo e morto.

4.2. Impacto do Tamanho do Tile no Desempenho do MPPA-256

O primeiro experimento tem por objetivo verificar o impacto do tamanho dos *tiles* no desempenho e consumo energético das aplicações. As Figuras 4 e 5 mostram, respectivamente, os tempos de execução e consumo de energia de três aplicações estêncil, variandose o tamanho do Array2D de entrada (de 2048^2 até 12288^2) e os tamanhos do *tile* (de 32^2 até 128^2). Arrays de entrada maiores que 12288^2 e *tiles* maiores que 128^2 extrapolam às memórias LPDDR3 e dos *clusters*, respectivamente.

¹http://ccl.northwestern.edu/netlogo/models/Fur



Figura 4. Tempos de execução das aplicações para diferentes tamanhos de *tile* e Array2D no MPPA-256.



Figura 5. Consumo de energia das aplicações para diferentes tamanhos de *tile* e Array2D no MPPA-256.

Pode-se perceber uma redução no tempo de execução à medida em que se aumenta o tamanho do *tile* (Figura 4), pois há menos sincronizações e comunicações de *tiles* entre os processos mestre e escravos. O comportamento das aplicações é similar, sendo diferenciado apenas pela grandeza dos tempos de execução. A Figura 5 apresenta um comportamento similar para o consumo de energia, pois o tempo de execução reduz com o aumento do *tile*, trazendo uma redução no consumo de energia.

4.3. Análise de Escalabilidade no MPPA-256

Em um segundo experimento, buscou-se verificar a escalabilidade das aplicações no MPPA-256. Para isso, variou-se o número de *clusters* em cada aplicação, com Array2D de entrada fixo de tamanho 4096^2 e *tiles* de tamanho 128^2 . A Figura 6(a) apresenta os tempos de execução obtidos ao variar-se o número de *clusters* utilizados na computação. A Figura 6(b), por outro lado, apresenta o fator de aceleração (*speedup*) com relação ao tempo de execução com 1 *cluster*. Em outras palavras, o *speedup* com *c clusters* é computado dividindo-se o tempo de execução obtido com apenas 1 *cluster* pelo tempo de execução obtido com *c clusters*.

No geral, os resultados mostraram que a solução proposta para o MPPA-256 é escalável. Porém, pode-se notar que a aplicação *Fur* apresentou uma escalabilidade superior às demais aplicações. Esse comportamento está diretamente relacionado com a quantidade de operações realizadas pelo *kernel* da aplicação (complexidade do *kernel*). Tendo em vista a necessidade de comunicações no MPPA-256, o tempo total de execução de uma aplicação passa a ser composto pela soma do tempo de comunicação com o tempo de computação. Para um dado *tile t* de tamanho fixo, o tempo necessário para realizar comunicações de *t* entre mestre e escravo será constante. Por outro lado, quanto maior o número de operações (computações) feitas em *t* pelo *kernel* da aplicação, maior será o paralelismo a ser explorado. Nesse caso, o tempo de computação será proporcionalmente maior que o tempo de comunicação, melhorando assim a escalabilidade obtida. Este é o caso da aplicação *Fur* cujo *speedup* se aproxima do caso ideal. Em contrapartida,



Figura 6. Resultados de tempo e speedup das aplicações Fur, GoL e Jacobi.

aplicação *Jacobi* apresentou uma escalabilidade mais baixa que as demais, pois seu *kernel* apresenta baixa complexidade.

4.4. Kalray MPPA-256 vs. Intel Broadwell

Por fim, foram efetuados experimentos comparativos entre o processador Intel Xeon e o MPPA-256, como pode-se ver na Figura 7. Nesses experimentos, utilizou-se um Array2D de entrada de tamanho 12288^2 e *tiles* de tamanho 128^2 . Ao ser comparado o tempo das aplicações em cada arquitetura, nota-se que o MPPA-256 tem um desempenho pior, contudo ao ser comparado o consumo de energia percebe-se um comportamento diferente: a energia consumida pelo MPPA-256 é menor, principalmente na aplicação *Fur*. No geral, o consumo de energia das aplicações *Fur*, *GoL* e *Jacobi* no MPPA-256 foi aproximadamente 1.45x, 1.38x e 1.27x menor que no Intel Xeon, respectivamente. Por outro lado, o tempo de execução dessas aplicações obtido no MPPA-256 foi 3.30x, 2.83x e 2.69x maior que no Intel Xeon, respectivamente.

5. Trabalhos Relacionados

Alguns trabalhos surgiram recentemente com intuito de avaliar o uso de processadores *manycore* em CAD, além de discutir os desafios do desenvolvimento de aplicações para esses processadores. Em [Totoni et al. 2012], os autores compararam o desempenho e o consumo energético de um processador *manycore* experimental da Intel denominado *Single-Chip Cloud Computer* (SCC) com outros tipos de processadores e GPUs. Para realizar essa análise, os autores utilizaram um conjunto de aplicações paralelas implementadas em Charm++ [Kale and Bhatele 2012]. Os resultados obtidos com o Intel SCC mostraram que *manycores* são uma alternativa viável, apresentando bom desempenho e baixo consumo energético. Em [Sirdey et al. 2013], os autores avaliaram o desempenho do processador *manycore* MPPA-256 no contexto de aplicações de decodificação de vídeo. Os resultados mostraram que o desempenho do MPPA-256 é comparável ao desempenho de processadores Intel atuais em uma decodificação de vídeo no padrão H.264, consumindo 6 vezes menos energia.

Trabalhos recentes revelaram o desempenho e consumo energético do processador MPPA-256, comparando-o a outros processadores *multicore* de propósito geral e embarcados, no contexto de diferentes aplicações científicas [Castro et al. 2014, Castro et al. 2013, Francesquini et al. 2014]. Os resultados mostraram que o processador *manycore* MPPA-256 apresenta em alguns casos desempenho superior a processadores *multicore* Intel Xeon 2.4 GHz com 8 *cores*, além de um consumo de energia de até 13 vezes menor em relação ao mesmo processador. Um outro trabalho recentemente publicado



Figura 7. Comparação do tempo de execução e consumo de energia das aplicações *Fur*, *GoL* e *Jacobi* em relação a arquitetura.

realizou uma análise comparativa de desempenho e consumo de energia entre processadores *multicore* Intel de alto desempenho e ARM [Padoin et al. 2015]. Os resultados mostraram que, apesar da potência dos processadores ARM ser pelo menos 10 vezes menor que a dos processadores Intel de alto desempenho, o consumo de energia nem sempre será melhor, sendo dependente das características da carga de trabalho a ser executada.

Trabalhos recentes que propuseram APIs e ambientes de execução para *manycores*. Em [Lam et al. 2013], os autores propuseram uma adaptação Espaço de Endereçamento Global Particionado (*Partitioned Global Address Space* – PGAS) para simplificar o desenvolvimento de aplicações paralelas para os processadores *manycore* TILE-Gx e TILEPro. Mais precisamente, os autores utilizaram a biblioteca OpenSH-MEM como base para a proposta, utilizando-a como uma camada de abstração para as bibliotecas fornecidas pelo fabricante dos processadores. Com isso, aplicações atualmente implementadas utilizando a API OpenSHMEM podem ser executadas nos processadores *manycore* da linha TILE sem que haja a necessidade de modificações no código.

6. Conclusão

O desenvolvimento de aplicações que exploram o paralelismo em processadores *many-core* de baixa potência, tais como o MPPA-256, tornou-se muito importante, tendo em vista o aumento do consumo de energia de processadores de alto desempenho. Porém, o desenvolvimento de aplicações otimizadas nesses processadores é bastante desafiador devido a fatores importantes tais como a existência de um modelo de programação híbrido, capacidade limitada de memória no *chip*, ausência de coerência de *cache*, entre outros.

Neste artigo foi proposta uma adaptação de um *framework* para desenvolvimento de aplicações estêncil iterativas, denominado PSkel, para processador MPPA-256. A solução proposta permite esconder detalhes de baixo nível do MPPA-256, simplificando significativamente o desenvolvimento de aplicações estêncil nesse processador. Os resultados mostraram que a solução proposta apresenta boa escalabilidade. Além disso, foi observado uma redução significativa no tempo de execução e no consumo de energia das aplicações no MPPA-256 ao se utilizar a técnica de *tiling* trapezoidal. Isso se deve, principalmente, à redução do sobrecusto de comunicações e sincronizações de *tiles*.

A aplicação *Fur* apresentou os melhores resultados de escalabilidade dentre as 3 aplicações estudas, obtendo um *speedup* de 14x em relação à apenas um *cluster*. Analisando experimentos executados sobre a adaptação pôde-se perceber uma relação entre a quantidade de computação realizada pelo *kernel* da aplicação e o *speedup* obtido.

Por fim, experimentos comparativos entre o MPPA-256 e o processador Intel Broadwell mostraram que a solução proposta para o MPPA-256 apresenta uma eficiência energética superior apesar de um tempo de execução superior.

Como trabalhos futuros, pretende-se estudar formas de reduzir ainda mais os sobrecustos de comunicação através do uso de técnicas de *sofware prefetching*. Além disso, pretende-se realizar experimentos com outros *benchmarks* e aplicações que utilizam estruturas tridimensionais. Por fim, pretende-se realizar comparações de desempenho e consumo de energia com outros processadores embarcados.

Referências

- Castro, M., Dupros, F., Francesquini, E., Méhaut, J.-F., and Navaux, P. O. A. (2014). Energy efficient seismic wave propagation simulation on a low-power manycore processor. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 57–64, Paris, France. IEEE Computer Society.
- Castro, M., Francesquini, E., Dupros, F., Aochi, H., Navaux, P. O., and Méhaut, J.-F. (2016). Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Computing*, 54:108–120.
- Castro, M., Francesquini, E., Nguélé, T. M., and Méhaut, J.-F. (2013). Analysis of computing and energy performance of multicore, NUMA, and manycore platforms for an irregular application. In *Workshop on Irregular Applications: Architectures & Algorithms (IA^3)*, pages 5:1–5:8, Denver, EUA. ACM.
- Cole, M. (2004). Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406.
- Demmel, J. W. (1997). Applied numerical linear algebra. SIAM.
- Enmyren, J. and Kessler, C. W. (2010). SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA. ACM.
- Francesquini, E., Castro, M., Penna, P. H., Dupros, F., de Freitas, H. C., Navaux, P. O. A., and Méhaut, J.-F. (2014). On the energy efficiency and performance of irregular applications on multicore, NUMA and manycore platforms. *J. Parallel Distrib. Comput.*, 76:32–48.
- Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., Zhao, W., Yin, X., Hou, C., Zhang, C., Ge, W., Zhang, J., Wang, Y., Zhou, C., and Yang, G. (2016). The sunway taihulight supercomputer: system and applications. *SCIENCE CHINA Information Sciences*, 59(7):072001:1–072001:16.
- Gardner, M. (1970). Mathematical Games The Fantastic Combinations of John Conway's New Solitaire Game 'Life'. *Scientific American*, 223(3).
- Gonzalez, R. C. and Woods, R. E. (2006). *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc.
- Holewinski, J., Pouchet, L.-N., and Sadayappan, P. (2012). High-Performance Code Generation for Stencil Computations on GPU Architectures. In ACM ICS, pages 311–320.
- Kale, L. V. and Bhatele, A., editors (2012). *Parallel Science and Engineering Applications: The Charm++ Approach.* CRC Press, 1st edition.

- Lam, B. C., George, A. D., and Lam, H. (2013). TSHMEM: Shared-Memory Parallel Computing on Tilera Many-Core Processors. In *IEEE International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW)*, pages 325– 334, Cambridge, USA. IEEE Computer Society.
- Lutz, T., Fensch, C., and Cole, M. (2013). PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems. ACM Trans. Archit. Code Optim., 9(4):59:1–59:24.
- Meng, J. and Skadron, K. (2011). A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations. *International Journal of Parallel Programming*, 39(1):115–142.
- Padoin, E. L., Pilla, L. L., Castro, M., Boito, F. Z., Navaux, P. O. A., and Méhaut, J.-F. (2015). Performance/Energy Trade-off in Scientific Computing: The Case of ARM big.LITTLE and Intel Sandy Bridge. *IET Computers & Digital Techniques*.
- Pereira, A. D., Ramos, L., and Góes, L. F. W. (2015). PSkel: A stencil programming framework for cpu-gpu systems. *Concurrency and Computation: Practice and Experience*, 27(17):4938–4953.
- Rocha, R. C. O., Pereira, A. D., Ramos, L., and Góes, L. F. W. (2017). TOAST: Automatic tiling for iterative stencil computations on GPUs. *Concurrency and Computation: Practice and Experience*, 29(8):e4053.
- Sirdey, P. A., Beaucamps, P.-E., Blanc, F., Bobin, B., Carpov, S., Cudennec, L., David, V., Dore, P., Dubrulle, P., de Dinechin, B. D., François Galea, Goubier, T., and Harrand, M. (2013). Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor. In *International Conference on Computational Science (ICCS)*, volume 18, pages 1624–1633, Barcelona, Spain. Elsevier Science.
- Steuwer, M., Kegel, P., and Gorlatch, S. (2011). SkelCL A Portable Skeleton Library for High-Level GPU Programming. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1176–1182, Washington, DC, USA. IEEE Computer Society.
- Totoni, E., Behzad, B., Ghike, S., and Torrellas, J. (2012). Comparing the Power and Performance of Intel's SCC to State-of-the-Art CPUs and GPUs. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 78–87, New Brunswick, Canada. IEEE Computer Society.
- Varghese, A., Edwards, B., Mitra, G., and Rendell, A. P. (2014). Programming the Adapteva Epiphany 64-core network-on-chip coprocessor. In *International Parallel Distributed Processing Symposium Workshops (IPDPSW)*, pages 984–992, Phoenix, USA. IEEE Computer Society.
- Weaver, V. M., Johnson, M., Kasichayanula, K., Ralph, J., Luszczek, P., Terpstra, D., and Moore, S. (2012). Measuring energy and power with PAPI. In 2012 41st International Conference on Parallel Processing Workshops, pages 262–268.