UNIVERSIDADE FEDERAL DE SANTA CATARINA

BRUNO MARQUES DO NASCIMENTO

OTIMIZAÇÃO DO *FRAMEWORK* PSKEL PARA O PROCESSADOR *MANYCORE* MPPA-256

Florianópolis

2019

Bruno Marques do Nascimento

OTIMIZAÇÃO DO *FRAMEWORK* PSKEL PARA O PROCESSADOR *MANYCORE* MPPA-256

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Márcio Bastos Castro

Florianópolis

2019

Catalogação na fonte pela Biblioteca Universitária da Universidade Federal de Santa Catarina.

Arquivo compilado às 11:30h do dia 14 de julho de 2019.

Bruno Marques do Nascimento
OTIMIZAÇÃO DO *FRAMEWORK* PSKEL PARA O PROCESSADOR *MANYCORE* MPPA-256 : / Bruno Marques do Nascimento; Orientador, Prof. Dr. Márcio Bastos Castro; , - Florianópolis, 11:30, 14 de julho de 2019.
87 p.

Trabalho de Conclusão de Curso - Universidade Federal de Santa Catarina, Departamento de Informática e Estatística, Centro Tecnológico, Curso de Bacharelado em Ciência da Computação.

Inclui referências

1. *Manycore*. 2. MPPA-256. 3. Processamento de alto desempenho. 4. Eficiência energética. I. Prof. Dr. Márcio Bastos Castro II. III. Curso de Bacharelado em Ciência da Computação IV. OTIMIZAÇÃO DO *FRAMEWORK* PSKEL PARA O PROCESSADOR *MANYCORE* MPPA-256

CDU 02:141:005.7

Bruno Marques do Nascimento

OTIMIZAÇÃO DO *FRAMEWORK* PSKEL PARA O PROCESSADOR *MANYCORE* MPPA-256

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciência da Computação, e aprovado em sua forma final pelo Curso de Bacharelado em Ciência da Computação do Departamento de Informática e Estatística, Centro Tecnológico da Universidade Federal de Santa Catarina.

Florianópolis, 14 de julho de 2019.

Prof. Dr. José Francisco Danilo de Guadalupe Correa Fletes
Coordenador do Curso de Bacharelado em Ciência da Computação

Banca Examinadora:

Prof. Dr. Márcio Bastos Castro
Orientador
Universidade Federal de Santa Catarina – UFSC

Prof. Dr. Frank Augusto Siqueira
Universidade Federal de Santa Catarina – UFSC

Prof. Dr. Odorico Machado Mendizabal
Universidade Federal de Santa Catarina – UFSC

Este trabalho é dedicado aos meus pais, que se privaram de luxos da vida para permitir que seus filhos tivessem uma educação de qualidade e não mediram esforços para que eu realizasse meus sonhos. À minha irmã, que me mostra que a paixão pelo que se exerce é imprescindível. Ao meu afilhado Miguel, que me fez sentir o que é ter uma extensão do meu corpo no mundo. E à minha futura esposa, que me ensinou que o amor é a verdadeira razão da vida.

AGRADECIMENTOS

Aos meus pais por todo suporte, amor e dedicação, sempre priorizando uma educação de excelência para seus filhos. E que em mim depositaram toda sua confiança ao longo dessa jornada de 4 anos e meio.

À minha namorada Samara Zimmermann, que me mostrou a força que um coração puro e generoso possui. Foi meu pilar emocional, que me incentivou e ajudou a manter o foco nos momentos mais difíceis. E me mostrou que desistir é algo que não existe no seu vocabulário.

À Cecília, minha segunda mãe, um presente da vida, que está sempre a disposição e que eu amo incondicionalmente.

Aos meus amigos, que compartilham comigo a aventura do mundo acadêmico da graduação, as dificuldades, as felicidades e toda a cumplicidade envolvida. Uma história que começou no ensino médio e foi estendida para a universidade e prosseguirá para a vida.

Ao meu orientador Márcio Bastos Castro, seu profissionalismo e dedicação comprovou a existência de professores excepcionais e comprometidos com o objetivo de ensinar e produzir conhecimento científico. Sempre disposto e disponível, foi um exímio orientador. Você é minha referência profissional que levarei para a vida.

E aos colegas de curso, que trilharam essa jornada ao meu lado e através da cumplicidade e companheirismo transpomos as barreiras que emergiam no decorrer do caminho.

O presente trabalho foi realizado com o apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq - Brasil.

"Never limit yourself because of others' limited imagination; never limit others because of your own limited imagination."
Mae C. Jemison

RESUMO

Uma nova classe de *chips* altamente paralelos de baixo consumo energético que lidam com a restrição de energia foi desenvolvida. Os processadores Sunway SW26010 e Kalray MPPA-256 são exemplos deles, entregando mais de duzentos núcleos de processamento em um único *chip*. Apesar de apresentarem melhor eficiência energética do que os processadores *multicore* de propósito geral, características arquiteturais como a limitada quantidade de memória distribuída no *chip* torna o desenvolvimento de aplicações científicas paralelas eficientes uma tarefa desafiadora. Neste projeto foram propostas otimizações ao *framework* PSkelMPPA, que provê uma abstração única e de alto nível para programação estêncil no processador MPPA-256, eximindo os programadores de serem responsáveis pela tarefa de explicitamente lidar com a comunicação e com o modelo de programação paralela híbrida do MPPA-256.

Palavras-chaves: *Manycore*. MPPA-256. Processamento de alto desempenho. Eficiência energética.

ABSTRACT

A new class of highly parallel low-power chips that deal with a energy restriction was developed. The Sunway SW26010 and Kalray processors are some exemples of them, giving more than two hundred processing cores in a single low-power chip. Despite presenting a better energy efficiency than the general purpose multi core processors, the architects features such as the limited amount of distributed memory on the chip makes the development of efficient scientific applications a challenging task. In this term paper were proposed optimizations to the framework PSkel MPPA, which provides an unique, high-level abstraction for stencil programming in the MPPA-256 processor, exempting programmers from being responsible for the task of explicitly handling with communication and with the parallel hybrid programming model of the MPPA-256.

Keywords: *Manycore*. MPPA-256. High-performance computing. Efficient energy.

LISTA DE FIGURAS

LISTA DE CÓDIGOS–FONTE

LISTA DE ABREVIATURAS E SIGLAS

HPC	*High Performance Computing*
E/S	Entrada e Saída
NoC	*Network-on-Chip*
CPU	*Central Processing Unit*
GPU	*Graphics Processing Unit*
API	*Application Programming Interface*
IPC	*Inter-Process Communication*
LPDDR3	*Low Power Double Data Rate 3*
RAM	*Random Access Memory*
UMA	*Uniform Memory Access*
NUMA	*Nonuniform Memory Access*

SUMÁRIO

1 INTRODUÇÃO

Até a última década, o desempenho das arquiteturas utilizadas na área de *High Performance Computing* (HPC) tem sido quase exclusivamente quantificado pelo seu poder de processamento. No entanto, a eficiência energética está sendo considerada recentemente tão importante quanto o desempenho e tornou-se um aspecto crítico para o desenvolvimento de sistemas escaláveis (FRANCESQUINI et al., 2015). Portanto, a pesquisa e o desenvolvimento científico nesta área com enfoque na redução do consumo de energia têm se tornado de extrema importância para o avanço tecnológico.

Assim, é notório o surgimento de uma nova classe de processadores, os denominados processadores *manycore*, com mais de centenas de núcleos de processamento e de baixo consumo de energia, capazes de lidar com o paralelismo de dados e tarefas, como o MPPA-256 (CASTRO, M. B. et al., 2013). Apesar dos benefícios oriundos dos processadores *manycore*, o uso deles traz desafios para a programação de aplicações paralelas devido às suas características arquiteturais (CASTRO, Márcio et al., 2016). Uma das formas de simplificar o desenvolvimento de aplicações paralelas, abstraindo os detalhes arquiteturais e de programação de baixo nível, é através da utilização de padrões paralelos ou esqueletos algorítimicos (COLE, 2004).

Um destes padrões é denominado estêncil. Este padrão consiste em varrer os elementos de um dado *array* de entrada de n-dimensões, e modificar o valor de cada elemento com base nos valores dos elementos vizinhos, produzindo assim um *array* de saída de n-dimensões com valores modificados. Além disso, essa etapa de varredura e modificação de valores pode ser realizada iterativamente, na qual o *array* de saída de uma iteração, será o *array* de entrada da iteração seguinte. É válido destacar que muitos destes padrões estão presentes nas mais diversas áreas do conhecimento, como física, matemática, processamento de imagens, entre outros (HOLEWINSKI; POUCHET; SADAYAPPAN, 2012).

Dentre os *frameworks* propostos, o PSkel se destaca por prover uma abstração para ambientes heterogêneos de programação, que contam com a presença de processadores *multicore* e placas gráficas. Além disso, foram observados ganhos na performance média de até 76% quando comparado com aplicações paralelas que utilizavam apenas *Central Processing Unit* (CPU) (PEREIRA; RAMOS; GÓES, 2015).

Com a necessidade e importância de cunho científico do desenvolvimento de aplicações paralelas nos processadores *manycore*, foi proposta uma adaptação do *framework* PSkel para que ele dê suporte ao processador MPPA-256. Os resultados obtidos com o PSkel no MPPA-256 foram promissores, apresentando uma redução no consumo energético de aplicações estêncil quando comparado com execuções em um processador de alto desempenho Intel Broadwell (PODESTÁ JR. et al., 2017).

Todavia, foi observado que o tempo desperdiçado na comunicação das aplicações do MPPA-256 é elevado e impacta diretamente nos testes e experimentos realizados. A comunicação ocorre através de *Networks-on-Chip* (NoCs) de dados e de controle. Decorrente desta estrutura, o atraso de comunicação entre elementos fisicamente mais distantes na rede será maior do que elementos mais próximos.

Além disso, a atual *Application Programming Interface* (API) de comunicação utilizada no PSkel para o MPPA-256 é similar ao modelo clássico POSIX de baixo nível para *Inter-Process Communication* (IPC), o que dificulta o desenvolvimento de rotinas de comunicação e expõe essas rotinas a possíveis perdas de desempenho não trivialmente detectáveis. A nova API de comunicação assíncrona desenvolvida e disponibilizada pelos desenvolvedores do processador eleva o nível de abstração dessas rotinas de comunicação com implementações otimizadas para o processador, acarretando em maior robustez e desempenho na utilização da NoC. Com isso, nota-se a importância da revisão e estudo da implementação atual da adaptação do *framework* PSkel para o MPPA-256, visando encontrar espaços para otimização e ganho de desempenho que impactarão diretamente no gasto energético do processador, que já se mostrou promissor

quando comparado ao Intel Broadwell.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

À medida que se reduz o número de comunicações e sincronizações, através do aumento do número de elementos processados a cada iteração de uma aplicação de computação estêncil no MPPA-256, é observada a queda no tempo de sua execução e, por consequência, a redução do consumo de energia (PODESTÁ JR. et al., 2017). Uma vez identificado que os gastos de comunicação são fatores relevantes para a influência do consumo de energia e tempo de execução das aplicações, este *Trabalho de Conclusão de Curso* tem como objetivo geral propor otimizações na comunicação de dados para o *framework* PSkel adaptado para o MPPA-256, visando o ganho de desempenho e redução de consumo de energia pelas aplicações paralelas de padrão estêncil no MPPA-256.

1.1.2 Objetivos Específicos

Os objetivos específicos deste trabalho são:

1. Estudar a nova API de comunicação assíncrona disponibilizada pelo fabricante do processador e realizar as modificações de código necessárias no PSkel para fazer uso da mesma;
2. Propor e implementar otimizações na comunicação e na distribuição de dados entre os elementos de processamento, a fim de obter ganhos de desempenho e reduzir o consumo de energia;
3. Realizar experimentos com aplicações estêncil para medir o desempenho e a eficiência energética das otimizações propostas, comparando resultados obtidos com outros processadores *multicore* e processadores gráficos.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta a fundamentação teórica sobre sistemas com múltiplos processadores, processador *manycore* MPPA-256, padrão estêncil, *framework* PSkel e a adapatação PSkel-MPPA.

2.1 SISTEMAS COM MÚLTIPLOS PROCESSADORES

Tanenbaum (TANENBAUM; BOS, 2014) classifica sistemas com múltiplos processadores em dois grupos: os sistemas multiprocessadores e os multicomputadores. Nesta seção serão apresentados os principais conceitos dos sistemas multiprocessadores e multicomputadores.

2.1.1 Multiprocessadores

O multiprocessador é um sistema de computadores no qual qual duas ou mais *Central Processing Units* (CPUs) compartilham acesso total a uma *Random Access Memory* (RAM). Assim, programas executando em qualquer uma das CPUs realizam operações de leitura e escrita em um espaço de endereçamento físico único na RAM. A particularidade que este sistema apresenta é que, ao escrever um valor na memória, posteriormente ao ler este mesmo valor ele pode obter um resultado diferente daquele que foi escrito, pois outra CPU pode alterá-lo. Percebe-se que este comportamento pode causar inconsistências no sistema e, por isso, é de extrema importância que recebam o tratamento adequado que, quando realizado, constitui a base para a comunicação entre processadores.

Apesar da propriedade de endereçamento total da memória, alguns multiprocessadores possuem a propriedade adicional de que cada palavra da memória pode ser lida tão rápido quanto a leitura de qualquer outra palavra presente na memória. Essas máquinas são denominadas multiprocessadores com acesso uniforme à memória – *Uniform Memory Access* (UMA). Outrossim, existem os multiprocessadores com acesso não uniforme à memória – *Nonuniform Memory Access* (NUMA).

2.1.1.1 Multiprocessadores UMA

Os multiprocessadores UMA possuem variações nas suas arquiteturas. Os mais simples são os baseados em barramento, no qual o barramento é utilizado para comunicação. Desta maneira, quando uma CPU deseja ler uma palavra da memória, será verificada a disponibilidade do barramento. Estando disponível, a CPU informa o endereço no barramento e junto de alguns sinais de controle aguarda até que a memória coloque a palavra requisitada no barramento. Porém, quando o barramento está indisponível, a CPU que deseja realizar leitura ou escrita da memória deve esperar até que o mesmo fique disponível. Quando existem poucas CPUs é possível realizar este gerenciamento, porém, quando o número de CPUs aumenta surge o problema, pois o sistema passa a possuir um gargalo no barramento, o que deixará muitas CPUs ociosas por muito tempo.

Uma maneira de reduzir o gargalo no barramento é através do uso de *caches*/memória. Dessa maneira requisições satisfeitas pela *cache* local não precisam ser realizadas através do barramento, desafogando o tráfego no mesmo e possibilitando o suporte a mais CPUs no sistema. Com o uso de *caches* surge também a necessidade de estabelecer a coerência entre as *caches* presentes no sistema. Para isso, são utilizados protocolos de coerência de *cache*, que serão responsáveis por manter a memória e as *caches* em um estado consistente.

2.1.1.2 Multiprocessadores NUMA

Mesmo com o melhor sistema de *cache*, o uso de um único barramento limita o tamanho de um multiprocessador UMA para cerca de 16 ou 32 CPUs (TANENBAUM; BOS, 2014). Para aumentar o número de CPUs, é necessário ir além e utilizar arquiteturas como barramento cruzado ou redes de comutação multi-estágio. Porém, estas arquiteturas necessitam de muito hardware, são caras, e não são tão maiores que as de barramento único. Para atingir a casa das dezenas de CPUs é necessário dispensar algo, normalmente o que é dispensado é o tempo de acesso uniforme para todos os módulos de memória. Isso resulta na ideia de multiprocessadores NUMA,

que, assim como nos UMA, o espaço de endereçamento é único entre as CPUs, porém o tempo de acesso aos módulos de memória locais é mais rápido que o acesso aos módulos remotos.

Existem dois grupos entre os NUMA, os que fazem uso da coerência de cache *Cache-Coherent NUMA* (CC-NUMA) e os que não fazem *Non Cache-coherent NUMA* (NC-NUMA). Os sistemas NUMA possuem nós compostos por CPU, *cache*, dispositivos de E/S e estão conectados por uma rede de interconexão.

2.1.1.3 *Chips* multinúcleo (*multicore*)

Os avanços tecnológicos na fabricação de chips resultam em transistores cada vez menores, possibilitando o uso de um número maior destes em único *chip*. Com tamanha quantidade de transistores em um único *chip* é possível explorar os megabytes de *cache* neles. Entretanto, existem limitações que fazem com que em um determinado ponto o aumento da *cache* gere um ganho de apenas 0,5% na taxa acerto (TANENBAUM; BOS, 2014). Outra possibilidade é colocar múltiplas CPUs completas, denominadas normalmente de núcleos, em um mesmo *chip*. Já são comuns *chips* com até 8 núcleos e a presença de *caches*, que são cruciais para o sistema, espalhadas pelo *chip*.

Chips multinúcleo são multiprocessadores muito pequenos, também são chamados de multiprocessadores em *chip* – *Chip Multi-Processors* (CMPs). Estes *chips* são semelhantes aos multiprocessadores baseados em barramento ou redes de comunicação. O que os difere destes, é o fato das CPUs serem conectadas muito próximas. Assim, falhas em componentes compartilhados podem acarretar na indisponibilidade de várias CPUs ao mesmo tempo, fato este que é improvável de ocorrer nos multiprocessadores tradicionais.

Também existem os sistemas em *chip* – *System on a Chip* (SoC), compostos de CPUs principais para processamento geral e de núcleos para tarefas específicas, como decodificação de aúdio e vídeo, criptoprocessadores, entre outros (TANENBAUM; BOS, 2014).

2.1.1.4 *Chips* com muitos núcleos (*manycore*)

Os *manycore* são *multicore* que contêm dezenas, centenas ou até milhares de núcleos. Embora não exista uma delimitação clara de quando um *multicore* torna-se um *manycore*, uma analogia utilizada é que um *multicore* é um *manycore* a partir do momento que a perda de um ou dois núcleos seja irrelevante (TANENBAUM; BOS, 2014).

A presença de um elevado número de núcleos em um mesmo *chip* acarreta em impactos sobre a coerência de *cache*, principalmente quanto ao custo e complexidade dos protocolos de coerência, que atingem um limite a partir do qual ele passa a prejudicar o desempenho do sistema em questão. Devido a isso, processadores *manycore* tendem a não utilizar *caches* coerentes (TANENBAUM; BOS, 2014).

A presença de milhares de núcleos não é mais algo tão incomum. As unidades de processamento gráfico – *Graphic Processing Unit* (GPU), presentes em muitos sistemas computacionais de hoje em dia, são detentores de milhares de pequenos núcleos com enfoque no processamento de dados, e realizam muito menos operações de coerência de *cache* e lógicas de controle que as CPUs de propósito geral. Estas mudanças estão refletidas na maneira de se programar para esses sistemas que, a exemplo das GPUs, utilizam linguagens específicas de programação, como *Open Computing Language* (OpenCL) e CUDA, além de abordar a ideia de "dados múltiplos e instrução única", ou seja, a execução de uma mesma instrução de máquina sobre diferentes fragmentos de dados.

2.1.2 Multicomputadores

Devido ao alto custo decorrente da dificuldade de se produzir grandes multiprocessadores surgiram os multicomputadores, que são CPUs propriamente acopladas e que não compartilham memória. Estes são fáceis de se construir, já que necessita apenas de um computador sem periféricos munido de uma placa de interface de rede de alto desempenho. O desempenho do sistema será determinado pela inteligência do projeto de interconexão das CPUs e suas placas de *interface*.

A necessidade de interconexão via rede implica em possibilidades de disposição dos nodos na rede, ou seja, possíveis topologias de rede conforme necessidade do sistema em questão, sendo elas topologia em anel, grade ou malha (e sua variante toro duplo – *torus 2D*), topologia em cubo e hipercubo.

Na topologia em anel os nodos estão conecatados diretamente a dois outros nodos da rede, sendo um elemento à direita e um à esquerda. Nesta topologia a existência de um comutador é dispensada.

A topologia em grade ou malha, é constituída de uma malha bidimensional que interliga diversos nodos e possui uma distância máxima possível conhecida para percorrer um caminho de um nodo origem para um nodo destino. Uma variação desta topologia é a toro duplo, que possui estrutura semelhante à topologia em grade. Porém, suas margens são interconectadas, possibilitando com que margens opostas consigam se comunicar diretamente sem precisar passar por todos o nodos intermediários.

Além disso, existem as topologias tridimensionais, representadas pelo formato de cubo devido à forma que assumem. A partir desta pode-se alcançar topologias n-dimensionais bastanto replicar a topologia de uma dimensão de tamanho $n - 1$ e ligar os nodos correspondentes, denominando-se topologia de hipercubo.

A comunicação em multicomputadores é realizada através da troca de mensagens utilizando rescursos disponibilizados pelo sistema operacional que podem ser reduzidos minimamente a duas chamadas de biblioteca, uma para enviar mensagens e outra para receber mensagens. A implementação destas bibliotecas de troca de mensagens possuem diversas variações de sistema para sistema.

2.2 MPPA-256

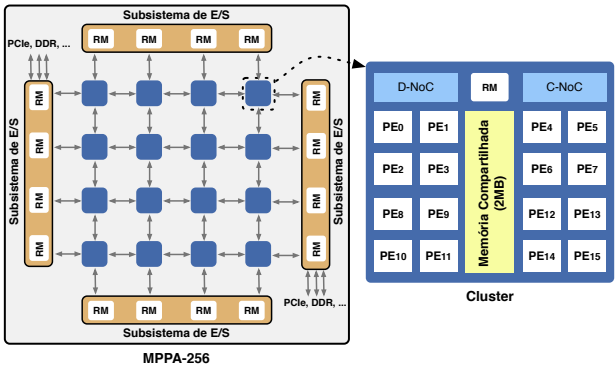
O MPPA-256 é um processador *manycore* desenvolvido pela empresa francesa Kalray. Apesar de ser considerado um *manycore* ele apresenta características de multicomputador, pois seus núcleos estão conectados através de uma NoC.

Esse processador possui 256 núcleos de usuário e 32 núcleos de sistema para processamento a 400 MHz. Esses núcleos estão distribuídos entre 16 *clusters* de computação e 4 *clusters* de *Input/Output*(E/S), que se comunicam através de NoCs de dados e controle. O processador utilizado no desenvolver deste projeto possui uma memória global de baixa potência (LPDDR3) de 2GB conectada a um dos subsistemas de E/S. A arquitetura do MPPA-256 é ilustrada na Figura 1. Cada cluster de computação tem os seguintes componentes:

- 16 núcleos chamados de *Processing Elements* (PEs), que são responsáveis por executar as *threads* de usuário (uma *thread* por *Processing Element*) e não pode ser interrompida ou preemptada;
- um *Resource Manager* (RM), responsável por executar o sistema operacional e gerenciar a comunicação;
- uma memória compartilhada de baixa latência de 2MB, que permite uma grande banda e fluxo de dados e controle entre os *Processing Elements* presentes no mesmo *cluster* de computação; e
- dois controladores de NoC, um para dados e outro para controle.

O processador apresenta um modelo de memória distribuída. Os *clusters* de computação e os *clusters* de E/S possuem espaço de endereçamento próprio. A exploração da computação paralela é realizada através de bibliotecas de código aberto, POSIX Threads (Pthreads) e OpenMP, e bibliotecas proprietárias, uma de baixo nível similar a POSIX IPC e a ASYNC, uma nova biblioteca disponibilizada pela Kalray e desenvolvida a partir da IPC. As primeiras, visam o paralelismo de computação nos *clusters* de computação através de memória compartilhada. Enquanto as últimas, seguem o modelo de memória distribuída e devem ser usadas para comunicação *cluster*–*cluster* e *cluster*–E/S via NoC.

Figura 1 – Arquitetura do MPPA-256.



Fonte: (CASTRO, M. B. et al., 2013)

A ASYNC é baseada em comunicação unilateral entre a memória local dos *clusters* de computação e a LPDDR3. Os principais conceitos por trás da ASYNC são domínios de execução, segmentos e operações de escrita e leitura. O domínio de execução representa um conjunto de núcleos compartilhando uma memória local, estando isolado de outros domínios de execução. Considerando o modelo de memória distribuída do MPPA-256, um domínio de execução corresponde a um *cluster* de computação ou a um *cluster* de E/S. Memória que não é acessível diretamente a partir dos núcleos de um domínio de execução pode ser estruturada em segmentos, que correspondem a toda ou parte da memória local de núcleos localizados em outro domínio de execução. Cada segmento tem uma assinatura única, que é especificada quando o segmento é criado em um domínio de execução através da função `mppa_async_segment_create()`. Então, outros domínios de execução podem referenciar um segmento previamente criado fornecendo a assinatura única para a função `mppa_async_segment_clone()`. Assim que segmentos são criados e referenciados por diferentes domínios de execução, eles podem realizar operações de escrita de dados da memória local para um segmento remoto ou leitura de um segmento remoto para a memória local. Diferentes funções para realizar estas operações estão disponíveis na biblioteca ASYNC, permitindo transferências de

dados contíguos ou espaçados (por exemplo, `mppa_async_put()` e `mppa_async_get_spaced()`), assim como, transferências de blocos 2D (`mppa_async_sget_block2d()`), que são úteis para transferir dados armazenados em estruturas bidimensionais.

O fluxo de execução de uma aplicação no MPPA-256 ocorre da seguinte maneira: o processo principal (chamado processo mestre) executa em um *Resource Manager* do *cluster* de E/S conectado à LPDDR3 e é responsável por alocar o dado de entrada na sua memória local (LPDDR3) e inicializar os processos trabalhadores (uma para cada *cluster* de computação) através da função `mppa_power_base_spawn()`. Os segmentos de dados necessários devem ser criados pelo processo mestre para que a transferência de dados com os *clusters* de computação aconteça. Por fim, o processo mestre deve esperar todos os trabalhadores terminarem através da função `mppa_power_base_waitpid()`. Cada processo trabalhador deve referenciar o segmento remoto criado na LPDDR3 para realizar leitura e escrita de dados durante a execução, e pode criar até 16 *threads* utilizando Pthread ou OpenMP (uma *thread* para cada *Processing Element*) para computar paralelamente. Cada *Processing Element* tem sua *cache* de memória privada sem qualquer mecanismo de coerência automático entre as outras *caches* de memória dos *Processing Elements* restantes. Embora isso melhore o desempenho da *cache*, é necessário que o desenvolvedor realize o *flush* explícito dos dados quando necessário.

A biblioteca IPC e seus detalhes serão retratados na seção 2.5, que aborda a atual implementação do PSkel-MPPA, que faz uso desta biblioteca.

Trabalhos anteriores mostraram que desenvolver aplicações paralelas otimizadas para o MPPA-256 é um grande desafio (FRANCESQUINI et al., 2015) devido a alguns fatores importantes, tais como: o modelo de memória distribuída presente no MPPA-256, a capacidade de memória dentro do *chip* e a comunicação explícita através da NoC. *Podestá Jr. et al.* (2017) apresentam mais detalhes sobre esses desafios.

2.3 ESQUELETOS PARALELOS

Um esqueleto paralelo é responsável por abstrair um determinado padrão de computação paralela. Para utilizar um esqueleto, o programador definirá as operações principais a serem realizadas, ou seja, definirá o *kernel* da computação. Esse esqueleto será responsável em compor essa função definida pelo usuário, devendo ser executada de maneira correta, paralela e o mais eficiente possível.

A abstração de padrões paralelos contribui para a simplificação e desenvolvimento, além de reduzir o custo de modelagem, facilitar a transformação e otimização das computações.

Devido ao alto nível de abstração, esqueletos paralelos possuem alta afinidade com conceitos presentes nas linguagens de programação, como *templates* e *generics* das linguagens de programação orientada a objetos. Assim, os esqueletos paralelos exploram estes mecanismos (GORLATCH; COLE, 2011).

2.3.1 Padrão estêncil

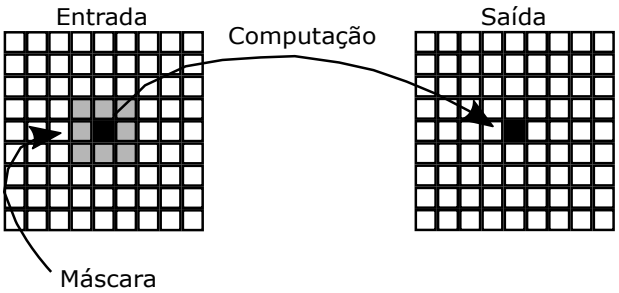
Dentre os padrões presentes na literatura, o padrão estêncil é muito utilizado devido a sua aplicação nos campos de simulações meteorológicas, comportamento de fluídos físicos, entre outros.

Ilustrado pela Figura 2, funciona da seguinte forma: para cada elemento de uma estrutura n-dimensional é computado um novo valor relativo aos vizinhos do elemento atual. Os vizinhos de um elemento são determinados pela máscara da computação. Por fim, cada novo valor computado é atribuído à sua célula respectiva em uma estrutura n-dimensional de saída. Em aplicações estêncil iterativas, a estrutura de saída é utilizada como estrutura de entrada de uma nova iteração da aplicação.

2.4 PSKEL

O PSkel é um *framework* de programação em alto nível para aplicações que utilizam o padrão estêncil, baseado no conceito de esqueletos paralelos, oferecendo suporte para a execução dessas a-

Figura 2 – O padrão estêncil.



Fonte: (PODESTÁ JR. et al., 2017)

plicações em ambientes heterogêneos, incluindo CPU e GPU. PSkel oferece um interface única de programação, desacoplada do *back-end* de execução, permitindo que o usuário se preocupe apenas em implementar o *kernel* estêncil que descreve a computação, enquanto o *framework* fica responsável pela tradução das abstrações descritas para código paralelo de baixo nível em C++, gestão de memória e transferência de dados. Tudo isso de forma transparente para o usuário (PEREIRA; RAMOS; GÓES, 2015).

O Código–fonte 1 mostra um exemplo do método Jacobi para resolver equações matriciais (DEMMEL, 1997) com o PSkel. A API do PSkel fornece estruturas genéricas para manipular os dados de entrada e saída, chamadas `Array`, `Array2D` (Código–fonte 1, linhas 12–13) e `Array3D`, através do uso de *templates* da linguagem de programação C++. Essas abstrações fornecem métodos que encapsulam o gerenciamento dos dados, como alocação de memória, cópia de memória e transferência de dados entre CPU e GPU. Além disso, fornece abstrações para definição da computação estêncil e gerenciamento de sua execução. A computação estêncil é definida na função `stencilKernel()` (Código–fonte 1, linhas 1–3), ela deve ser implementada pelo usuário do PSkel, é o método específico de cada aplicação, o qual descreve como a computação será realizada sobre cada elemento da matriz de entrada e seus vizinhos. O conjunto de classes para gerenciamento da execução são `Stencil`,

Código–fonte 1 – Trecho simplificado de código estêncil com PSkel.

```
1  __parallel__ void
2  stencilKernel(Array2D<float> A, Array2D<float> B,
3                struct Arguments args, int x, int y) {
4
5      B(x,y) = args.alpha * (A(x,y+1) + A(x,y-1) + A(x+1,y) + A(x-1,y)
6               + args.beta);
7  }
8
9  void jacobi(float *A, float *B, int M, int N, float alpha,
10             float beta, int timesteps) {
11
12     Array2D<float> input(A,M,N);
13     Array2D<float> output(B,M,N);
14     struct Arguments args(alpha, beta);
15     Stencil2D<Array2D<float>, struct Arguments>
           stencil(input,output,args);
16     stencil.runIterativeGPU(timesteps);
17 }
```

`Stencil2D` (Código–fonte 1, linha 15) e `Stencil3D`.

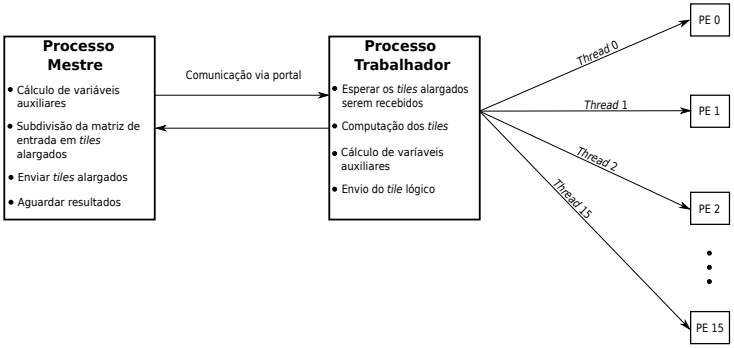
No exemplo fornecido, a função `stencilKernel()` será executada na GPU. O método `runIterativeGPU()` livra o usuário de escrever código específico na linguagem CUDA, necessário para excutar corretamente a computação estêncil na GPU.

2.5 PSKEL-MPPA

A adaptação PSkel-MPPA, é uma adaptação do PSkel proposta por (PODESTÁ JR. et al., 2017), ela faz uso de uma API similar à POSIX IPC para comunicação e será tratada como IPC no decorrer deste trabalho. Nela, são utilizados portais de comunicação para o envio de dados e o método de *strides* para gerenciar explicitamente o envio e recebimento de *tiles*.

A Figura 3 ilustra a adaptação desenvolvida. É empregado o modelo mestre-trabalhador, que é um dos padrões de computação paralela utilizado quando existem múltiplos núcleos de processamento. O processo mestre é executado no *cluster* de E/S conectado a memória LPDDR3, aonde os dados de entrada e saída (os *Array2Ds*) são alocados. Em seguida, ele calcula o número de *tiles* alargados que serão produzidos, assim como suas dimensões baseado em: i) parametros definidos pelo usuário, como o tamanho da entrada de dados

Figura 3 – Esquemático da proposta PSkel-MPPA.



Fonte - (PODESTÁ JUNIOR, 2018)

e as dimensões do *tile* lógico, o número de *clusters* de computação e o número de iterações internas; e ii) parâmetros do *kernel* estêncil, como o tamanho da máscara. Então, são lançados até 16 processos trabalhadores (um em cada *cluster* de computação) e é informado a cada processo trabalhador, o *tile* pelo qual será responsável por processar na atual iteração. Cada processo trabalhador, por outro lado, aloca dados para armazenar o *tile* alargado de entrada e de saída na memória local do *cluster* de computação.

A fase de computação consiste da execução do *kernel* estêncil pelos processos trabalhadores. As seguintes três principais etapas são realizadas para computar cada *tile* atribuído a um processo trabalhador: 1) o *tile* alargado é extraido do dado de entrada alocado na LPDDR3 e transferido para a memória local do *cluster* de computação para ser processado, a extração e transferência é realizada com o uso de portais; 2) t' iterações do *kernel* estêncil (iterações internas) são executadas pelo processo trabalhor sobre o *tile* alargado, sincronizando com o processo mestre ao final das iterações internas, com o objetivo de receber um novo *tile* ou para preparação dos dados a serem computados pelos trabalhadores nas iterações seguintes, ou término da execução; e 3). Ao alcançar o número total de iterações

(t) definido pelo usuário, estará presente na LPDDR3 o *Array2D* computado.

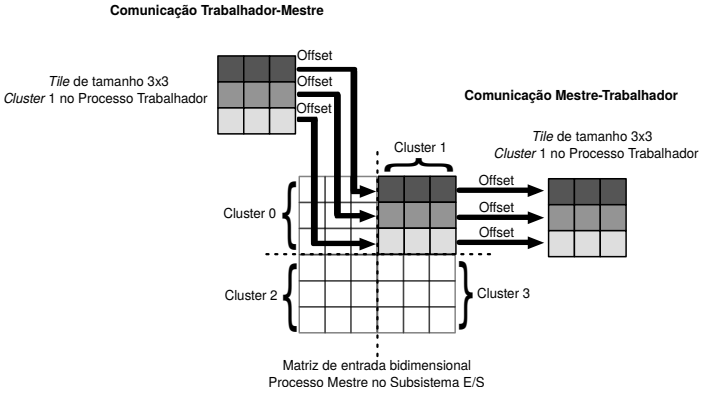
A comunicação realizada entre os processos mestre e trabalhadores é realizada por meio de portais, provenientes de uma API proprietária de baixo nível do MPPA-256, similar à POSIX IPC. Os portais são responsáveis por realizar escrita e leitura de maneira remota, para isso um processo que deseja comunicar-se deve criar um portal com uma classificação (*tag*) e relacioná-lo a um espaço de endereçamento, para posteriormente realizar leitura ou escrita no endereço associado.

A memória nos *clusters* de computação (2MB) é limitada, devido a isso a matriz de entrada da computação (*Array2D*) é particionada em tamanhos menores denominados *tiles*, que possuem tamanho fixo definido pelo usuário para serem enviados aos *clusters*.

As operações sobre *tiles* e matrizes são realizadas sobre endereços de memória. Devido a restrição da API e da NoC no MPPA-256, os dados armazenados em cada *tile* precisam estar contíguos em memória para serem transferidos pela NoC. Com o objetivo de evitar cópias locais de dados, utiliza-se o conceito de *strides*. Cada *stride* é uma parte contígua do `Array` original, sendo determinado por deslocamentos (*offsets*) especificados durante a execução. Os deslocamentos são dinâmicos e dependem dos *tiles* sendo computados. Essas informações são conhecidas pelo processo mestre por meio da utilização da classe `StencilTiling`, e este as repassa aos processos trabalhadores (PODESTÁ JUNIOR, 2018).

A Figura 4 ilustra o processo de comunicação com o método *strides* para o caso de uma matriz de entrada de tamanho 6x6, *tiles* de tamanho 3x3 e 4 *clusters*. Ao computar todos os deslocamentos, o método irá enviar diretamente, via portal, os dados especificados pelo endereço inicial do *tile* e pelos saltos na memória com a área útil especificada. Com isso, é possível enviar de forma contígua e direta os *tiles* a outro processo, sem a necessidade de cópias locais (PODESTÁ JUNIOR, 2018).

Figura 4 – Exemplo do funcionamento do método *strides* no MPPA-256.

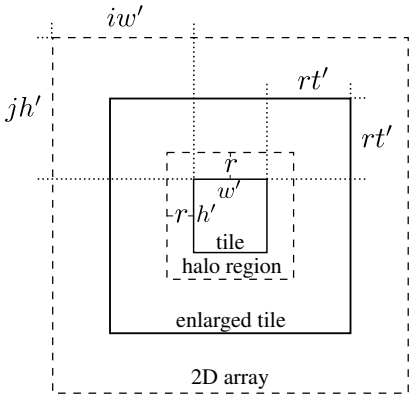


Fonte: (PODESTÁ JUNIOR, 2018)

Quando se trata de particionamento de computação estêncil, é necessário tratar as depedências de vizinhança provenientes do padrão paralelo estêncil, antes de particionar os dados de entrada. A técnica de *tiling* trapezoidal é utilizada para tratar as dependências de vizinhança, o que resulta na computação e presença redundante de dados (ROCHA et al., 2017), porém em troca da presença de computação e dados redundantes, se reduz a quantidade de comunicações e sincronizações necessárias a serem realizadas, que se mostram extremamente prejudiciais para o desempenho. A definição formal está ilustrada a seguir. Seja A uma matriz de dados 2D, com dimensões $\dim(A) = (w, h)$, aonde w e h são, respectivamente, sua largura e altura. Utilizando *tiles* de dimensões (w', h') temos $\lceil \frac{w}{w'} \rceil \lceil \frac{h}{h'} \rceil$ como sendo os possíves *tiles* de A. Seja $A_{i,j}$ um único *tile*, onde $0 \leq i < \lceil \frac{w}{w'} \rceil$ e $0 \leq j < \lceil \frac{h}{h'} \rceil$. $A_{i,j}$ possui um deslocamento de (iw', jh') em relação ao canto superior esquerdo de A e $\dim(A_{i,j}) = (\min\{w', w - iw'\}, \min\{h', h - jh'\})$. Esse deslocamento atua como um indexador necessário para acessar os elementos de um *tile*.

A Figura 5 mostra a representação gráfica dessa técnica. Um

Figura 5 – Técnica de *tiling* 2D.



Fonte: (ROCHA et al., 2017)

tile lógico (linha contínua interna) está contido em uma matriz de dados 2D (linha pontilhada externa) com deslocamento vertical e horizontal definidos por jh' e iw'. Se t iterações de uma aplicação estêncil devem ser executadas, é possível computar t' consecutivas iterações em $A_{i,j}$ ($t' \in [1, t]$) se precisar de qualquer troca de dados entre *tiles* adjacentes (também conheçido como iterações internas). Assim, o *tile* lógico ($A_{i,j}$) precisa ser alargado, com o acréscimo de uma *ghost zone* (área entre a linha sólida interna e externa), o que inclui a *halo region* (a área entre a linha sólida interna e a linha pontilhada interna). Seja r o mais distante deslocamento necessário para a vizinhança definida pela máscara estêncil. A área de alcance r que define a vizinhança é denominada *halo region*. O número de *halo regions* adjacentes que compõe a *ghost zone* é proporcional a t'. Assim, *tile* alargado $A^*_{i,j}$ possui deslocamentos $(\max\{iw' - rt', 0\}, \max\{jh' - rt', 0\})$ relativos a A (PODESTÁ JUNIOR, 2018).

Felizmente, todas as complexas tarefas relacionas a técnica de *tiling*, comunicação via NoC e adaptações discutidas nessa seção são transparentes para o usuário, tendo em vista que estão incluidas no *back-end* do PSkelMPPA.

3 TRABALHOS RELACIONADOS

Devido a importância dos esqueletos paralelos, e especifcamente o padrão paralelo estêncil, muitos esforços de pesquisas recentes buscam melhorar o desempenho e o suporte desses esqueletos em processadores *manycore*. *Buono et al.* (2013) portaram um *framework* baseado em esqueletos paralelos, chamado *FastFlow*, para o processador *manycore* TilePro64, que possui 64 núcleos de processamento idênticos, interconectados por uma malha de NoC. Similarmente, *Thorarensen et al.* (2016) apresentaram um novo *back-end* do *framework* SkePU para o processador *manycore* Myriad2, que possui como característica uma arquitetura heterogênea, visando dispositivos com restrição de energia e principalmente aplicações de visão computacional. *Gysi et al.* (2015) propuseram um *framework* para otimização automática da repartição de computações estêncil em sistemas híbridos de CPU e GPU.

Recentes trabalhos estudaram o desempenho e/ou a eficiência energética de processadores *manycore* de baixa potência. *Totoni et al.* (2012) compararam a potência e o desempenho do *Intel's Single-Chip Cloud Computer* (SCC) com outros tipos de *CPUs* e *GPUs*. Porém, eles mostraram que não existe um solução única que entrega o melhor troca entre potência e performance, os resultados mostram que *manycores* são uma oportunidade para o futuro. *Souza et al.* (2016) propuseram um conjunto de *benchmarks* para avaliar o MPPA-256 *manycore* processor. O *benchmark* oferece diversas aplicações que utilizam padrões paralelos, tipos de trabalho, intensidade de comunicação e estratégias de carga de trabalho, adequado para uma ampla compreensão do desempenho e consumo de energia do MPPA-256 e novos *manycores* que estão por vir. *Francesquini et al.* (2015) avaliaram três diferentes classes de aplicação (consumo de CPU, consumo de memória e uma composição híbrida dos dois tipos anteriores) utilizando plataformas de alto paralelismo como o MPPA-256 em uma plataforma NUMA de 24 nós e 192 núcleos. Eles mostraram que as arquiteturas *manycore* podem ser competitivas, mesmo se a aplicação é irregular por natureza.

De acordo com relevante conhecimento na área, o PSkel-MPPA é a primeira implementação completa de um *framework* com uso de padrões paralelos no MPPA-256. A solução proposta (PODESTÁ JUNIOR, 2018) livra os programadores da necessidade de lidar explicitamente com a gestão de comunicação e envio de dados pela NoC, assim como a preocupação de lidar com um ambiente híbrido de execução e a ausência de coerência de *cache* no MPPA-256.

4 PROPOSTA E IMPLEMENTAÇÃO DE OTIMIZAÇÃO NO PSKEL-MPPA

Com a disponibilização da nova biblioteca de comunicação (ASYNC) pela empresa Kalray para o MPPA-256, realizou-se o estudo da estrutura e funcionamento da adaptação do PSkel-MPPA, visando modificações na implementação proposta para fazer uso da nova API e buscar otimizações anteriormente não exploradas, almejando a melhora do desempenho e de sua eficiência energética.

4.1 IMPLEMENTAÇÃO COM A NOVA API DE COMUNICAÇÃO ASYNC

O fluxo de execução do PSkel-MPPA continuou semelhante. Durante a fase de inicialização, o processo mestre que está executando no *cluster* de E/S aloca os dados de entrada e saída na LPDDR3, e cria um segmento (conceito da nova API) específico para cada um deles. Em seguida, ele calcula o número de *tiles* alargados que serão produzidos assim como suas dimensões baseado em: i) parâmetros definidos pelo usuário, como o tamanho da entrada de dados e as dimensões do *tile* lógico, o número de *clusters* de computação e o número de iterações internas; e ii) parâmetros do *kernel* estêncil, como o tamanho da máscara. Então, são lançados até 16 processos trabalhadores (um em cada *cluster* de computação) e é informado a cada processo trabalhador o número de *tiles* alargados gerados, suas dimensões e o subconjunto de *tiles* que cada *cluster* será responsável por processar, ou seja, ao receber os dados do processo mestre inicialmente o processo trabalhador sabe qual conjunto de *tiles* ele será responsável e como deverá computá-lo. Por fim, o processo mestre aguarda até que todos os trabalhadores terminem de computar. Cada processo trabalhador, por outro lado, aloca dados para armazenar os *tiles* alargados de entrada e de saída na memória local do *cluster* de computação e clona ambos os segmentos de entrada e saída que foram criados pelo processo mestre para realizar futuras transferências de dados. A fase de inicialização tanto do mestre como do trabalhador está encapsulada na classe `Stencil2D`. Essa

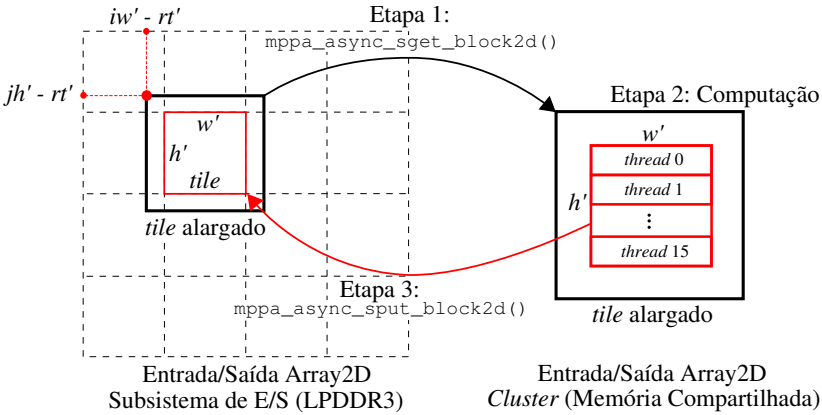
primeira etapa do fluxo de execução diferencia-se da anteriormente presente no PSkelMPPA, pois agora esta comunicação é realizada uma única vez na inicialização, o que não ocorria anteriormente, já que existiam trocas de mensagens de sincronização a cada laço da computação iterativa.

A fase de computação consiste da execução do *kernel* estêncil pelos processos trabalhadores. As seguintes três principais etapas são realizadas para computar cada *tile* atribuído a um processo trabalhador: 1) o *tile* alargado é extraído do dado de entrada alocado na LPDDR3 e transferido para a memória local do *cluster* de computação para ser processado; 2) t' iterações do *kernel* estêncil (iterações internas) são executadas pelo processo trabalhador sobre o *tile* alargado; e 3) o *tile* lógico resultante é transferido de volta da memória local do *cluster* de computação para a sua posição correspondente na LPDDR3. Assim que todos os *tiles* atribuídos a cada processo trabalhador forem computados com sucesso, todos os processos trabalhadores irão sincronizar em uma barreira global, pois os processos trabalhadores precisarão dos dados computados pelos outros durante as t' iterações para resolverem as dependências de vizinhança das próximas iterações a serem computadas. Foi usada a função `mppa_rpc_barrier_all()` (proveniente da API ASYNC) para esse propósito. Todo este processo anteriormente descrito é repetido até que o número total de iterações definido pelo usuário (t) seja alcançado.

As etapas acima mencionadas estão retratadas na Figura 6. Para sua implementação foi utilizada a nova API de comunicação assíncrona (ASYNC). Abaixo elas são descritas em mais detalhes:

Etapa 1. Baseado na informação fornecida pelo processo mestre durante o processo de lançamento dos *clusters* de computação, o processo trabalhador é capaz de calcular as coordenadas de cada *tile* alargado atribuído a ele em relação aos dados de entrada alocados na LPDDR3 (coordenadas $iw' - rt'$ e $jh' - rt'$) sem nenhuma intervenção extra do processo mestre. A função `mppa_async_sget_block2d()` recebe essa informação e o tamanho do bloco como parâmetros de entrada e ela transfere

Figura 6 – Comunicações com `block2d`.



Fonte: o autor.

o *tile* alargado para ser processado pelo processo trabalhador do segmento remoto de entrada para a memória local do *cluster* de computação através da NoC.

Etapa 2. O processo trabalhador computa as t' iterações do *kernel* estêncil definido pelo usuário sobre o *tile* alargado. Em cada t' iteração, a computação é paralelizada por meio de uma região paralela de OpenMP. A região paralela cria até 16 *threads* (uma para cada *Processing Element*). Cada *Processing Element* é responsável por executar o *kernel* estêncil em um subconjunto das células de um *tile* alargado, similar a versão com IPC.

Etapa 3. Após a computação do *kernel* estêncil, o *tile* lógico resultante é transferido de volta para a LPDDR3. A função `mppa_async_sput_block2d()` é usada para esse propósito, permitindo que o *tile* lógico seja extraído do *tile* alargado na memória local do *cluster* de computação e seja transferido para sua posicão correspondente no segmento de saída remoto, sem precisar se preocupar com a criação de portais e o uso do método de *strides*.

4.2 OTIMIZAÇÃO DA COMPUTAÇÃO DOS ELEMENTOS DE BORDA DA MATRIZ DE DADOS

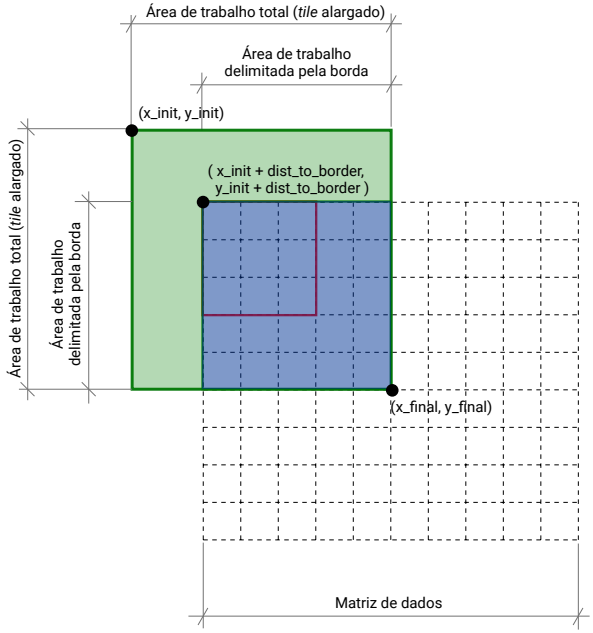
Ao realizar a computação de elementos próximos às bordas, é feita uma verificação em cada elemento para determinar se ele pertence à borda ou não e tratá-lo conforme necessário. Elementos de computação que beiram a borda física da matriz de entrada precisam acessar os elementos vizinhos, que neste caso são grande parte elementos inacessíveis fisicamente, o que pode acarretar em erros de acesso a memória, como a falha de segmentação. A implementação IPC realiza no processo de computação esta verificação para cada elemento do *tile* presente no *cluster* de computação, o que pode degradar seu desempenho. Com isso, pretende-se abordar uma nova maneira de tratamento para os elementos do *tile* que beiram a borda da matriz de dados.

A ideia é definir uma área de trabalho para cada *tile* que será computado e gerenciar essa área ao longo das iterações, reduzindo significativamente o número de verificações que ocorre em cada elemento do *tile* e em cada iteração. Na abordagem proposta neste TCC, o *cluster* já possui as coordenadas sobre as quais ele deve realizar sua computação, sem precisar verificar elemento por elemento. No caso dos *tiles* localizados centralmente na matriz sem contato com a borda, esta área de trabalho seria a máxima possível sem sofrer restrições. Esta área de trabalho é definida por uma *struct*, que é um tipo de dado composto na qual uma lista de variáveis podem ser declaradas para mais tarde serem acessadas através de um ponteiro único para cada uma. O Código–fonte 2 apresenta esta estrutura, e a Figura 7 ilustra seu uso.

Código–fonte 2 – Trecho de código que define a área de trabalho.

```
1 struct work_area_t {
2 int x_init,  /* X inicial da área de trabalho */
3     y_init,  /* Y inicial da área de trabalho */
4     x_final, /* X final da área de trabalho */
5     y_final; /* Y final da área de trabalho */
6     std::vector<int> dist_to_border; /* Vetor que define as
          distâncias entre os elementos mais externos e as bordas */
7 };
```

Figura 7 – Ilustração área de trabalho (*struct work_ area*).



Fonte: o autor.

Nesta estrutura são definias duas coordenadas, `(x_init, y_init)` e `(x_final,y_final)` que delimitam a área de trabalho. O vetor `dist_to_border` contém a distância entre os elementos mais externos e as bordas da matriz de dados, e com base nesses valores define os deslocamentos necessários a serem aplicados às coordenadas que delimitam a área de trabalho. Todos esses cálculos e definições encontram-se encapsulados dentro do *back-end* do PSkel-MPPA ASYNC, o desenvolvedor que faz uso do *framework* não precisa lidar com estes detalhes.

5 RESULTADOS

Este capítulo apresenta as plataformas de execução das aplicações, aplicações utilizadas, o estudo empírico do número de iterações internas (t'), o impacto do tamanho do *tile* no desempenho e os resultados da avaliação de desempenho e consumo energético obtidos, comparando-os com a versão IPC apresentada em (PODESTÁ JUNIOR, 2018) e com implementações disponíveis no PSkel para plataformas *multicore* (CPU) e GPU.

5.1 PLATAFORMAS

No MPPA-256, as medições referentes ao consumo de energia foram coletadas através de sensores disponíveis na plataforma, que englobam os *clusters*, a memória (LPDDR3 e memória local de cada *cluster*), subsistemas de E/S e a NoC. A compilação foi realizada utilizando GCC 5.4 (MPPA-256 e CPU) e NVCC versão 8.0 (GPU) com as *flags* de otimização `-O3` (todas as plataformas), `-march=native -mtune=native -ftree-vectorize` (CPU e GPU) e `-arch=sm_35` (GPU). As implementações para CPU e GPU foram executadas nas plataformas abaixo descritas.

- ***Xeon E5***: servidor *desktop* com processador Intel Xeon E5-2640 v4 (Broadwell) de 10 núcleos físicos executando a 2.4 GHz e 64 GB de RAM. As medições de energia nesta plataforma utilizaram a *interface RAPL* da Intel, que considera o consumo de energia de componentes de *hardware* através de contadores físicos. Esta *inteface* forneceu o consumo de energia da unidade de processamento e da memória.
- ***Tesla K40***: placa gráfica NVIDIA Tesla K40c com 2880 núcleos CUDA de processamento paralelo, executando a 745 MHz e com 12 GB de memória GDDR5. As medições de energia foram obtidas por intermédio da biblioteca de gerenciamento da NVIDIA (*NVML*). A *NVML* foi usada para obtenção do uso de energia da GPU e seus circuitos associados (*e.g.,* memória interna).

5.2 APLICAÇÕES

As aplicações utilizadas foram *Fur*, *GoL*, *Jacobi* e *Convolution*. Sendo as três primeiras utilizadas na comparação ASYNC *vs.* IPC e as três últimas na comparação ASYNC *vs.* GPU *vs.* CPU.

Foram utilizadas aplicações diferentes para os experimentos realizados, pois devido a uma atualização realizada no MPPA-256 uma das aplicações (FUR) mostrou desempenho inconsistente, os valores de tempo de execução obtidos pré-atualização divergiam de maneira incoerente após a atualização sem ter sido alterado nenhuma linha de código. Experimentos foram realizados visando entender o ocorrido, porém devido as restrições de tempo e informações limitadas fornecidas referente a esta atualização, optou-se pela troca da aplicação na comparação do MPPA-256 com outras plataformas.

- ***Fur***: é um modelo que implementa a maneira como os padrões de pele (listras das zebras, manchas dos leopardos e girafas) nos animais se auto-organiza. Foi primeiramente proposto por Alan Turing. A ideia é que esses padrões são formados individualmente em cada animal, e são influenciados por células de pigmentação presentes no pelo que irão influenciar outras células de pigmentação na sua vizinhança. Dessa maneira, cada animal possui um padrão diferente, mas como as células que irão definir esses padrões são herdadas dos pais, o desenvolvimento desses padrões faz com que os filhotes produzam padrões semelhantes com o de seus pais (WILENSKY, 2003). Assim, esta aplicação modelará a pele do animal em uma matriz bidimensional de células de pigmento que podem estar em um dos dois estados: colorida ou não-colorida. Esses estados serão influenciados pela presença de ativadores e inibidores produzidos pelas células da vizinhaça, no qual um nível mais alto de ativadores coloca a célula no estado colorida, um nível mais alto de inibidores leva a célula ao estado de não-colorida e o nível igual de inibidores e ativadores não muda o estado da célula.

- ***GoL (Game of Life)***: é um autômato celular que implemen-

ta o Jogo da Vida de Conway (GARDNER, 1970). O autômato é representado por uma matriz na qual cada célula representa um indivíduo no estado vivo ou morto. Com o passar das gerações (iterações), cada célula terá seu estado recalculado de acordo com sua vizinhança. Existem três possíveis ocorrências com o estado de cada célula. Primeiro, o estado permanece inalterado, quando a célula estiver morta e possuir um número de vizinhos vivos diferente de 3, ou a célula está viva e possuir 2 ou 3 vizinhos vivos. Segundo, o estado se altera de morto para vivo, quando uma célula morta possui exatamente 3 vizinhos vivos ("reprodução"). Terceiro, uma célula viva morre, quando a célula viva possui menos de 2 vizinhos vivos ("solidão") ou mais de 3 vizinhos vivos ("escassez de recursos para sobrevivência") (PEREIRA; RAMOS; GÓES, 2015).

- ***Jacobi***: Método iterativo para resolver sistemas de equações lineares (DEMMEL, 1997). O método converge garantidamente se a matriz de entrada é restrita ou irredutivelmente 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 $n \times n$.

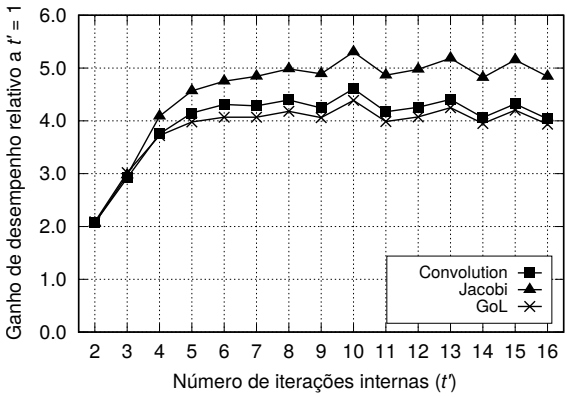
$$u'_{i,j} = \frac{u_{i\pm1,j} + u_{i,j\pm1} + h^2 f_{i,j}}{4} \tag{1}$$

  A cada passo, o novo valor de $u_{i,j}$ é obtido 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 (PODESTÁ JUNIOR, 2018).

- ***Convolution***: a convolução é uma técnica utilizada em diversas áreas de estudos. No processamento digital, é possível utilizá-la para realizar filtros de sinais, e assim atenuar ou realçar um áudio, uma imagem ou até mesmo um vídeo. O processo de convolução em uma imagem digital, recebe uma imagem I de largura W e altura H e a transforma em uma imagem O

de mesma dimensão. Uma matriz M de coeficientes é utilizada em conjunto com a imagem a ser processada para calcular a imagem resultante (PEREIRA; RAMOS; GÓES, 2015).

Figura 8 – Estudo empírico para encontrar o melhor valor para t'. Melhor *tradeoff* é alcançado com $t' = 10$.



Fonte: o autor.

5.3 ESTUDO EMPÍRICO: ITERAÇÕES INTERNAS (t')

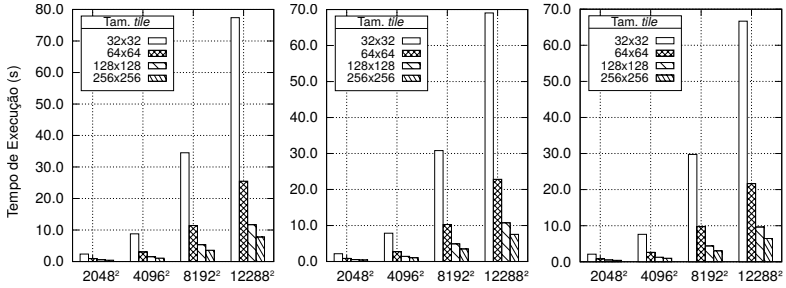
Devido ao uso da técnica de *tiling* trapezoidal, é possível definir com precisão o tamanho da *ghost zone* mediante o parâmetro t'. Essa é uma característica importante, que permite explorar a baixa quantidade de memória presente nos *clusters* e fazer melhor uso da NoC, pois a *ghost zone* impacta diretamente no tamanho e no consumo de memória do *tile* alargado. Assim, um estudo empírico com as aplicações supracitadas fora realizado para definir um valor ótimo para esse parâmetro. A Figura 8 expõe o ganho de desempenho quando variado o valor de t' no intervalo de 2 a 16 (o ganho de desempenho é em relação a $t' = 1$). Como já mencionado, existe um *tradeoff* entre o custo da computação redundante e a redução de comunicação e sincronização na NoC. O estudo empírico mostra que o melhor *tradeoff* é alcançado com $t' = 10$. Devido a isso, os resultados obtidos nas seções seguintes utilizam $t' = 10$.

5.4 TAMANHO DO *TILE VS.* DESEMPENHO

Nesta seção é analisado o impacto do tamanho do *tile* no desempenho das aplicações do PSkel no MPPA-256 com a versão ASYNC. Foram considerados quatro tamanhos de dados de entrada (2048x2048, 4096x4096, 8192x8192, 12288x12288) e quatro tamanhos de *tiles* (32x32, 64x64, 128x128, 256x256). Estes tamanhos máximos foram cuidadosamente selecionados para não extrapolar a quantidade de memória disponível para uso no MPPA-256 (2GB na LPDDR3 e 2MB em cada *cluster*). O número de iterações utilizado foi $t = 100$. Os resultados obtidos representam a média de 20 execuções com o desvio padrão médio menor que 1%.

A Figura 9 mostra o desempenho das aplicações estêncil quando variado o tamanho do dado de entrada e o tamanho do *tile*. Ao dobrar o tamanho da entrada observa-se um aumento médio de 2x a 3,3x no tempo de execução. Este comportamento é esperado, pois com o aumento da quantidade de dados mais computações, comunicações e sincronizações são necessárias.

Figura 9 – *Tiles* vs. tempo de execução.



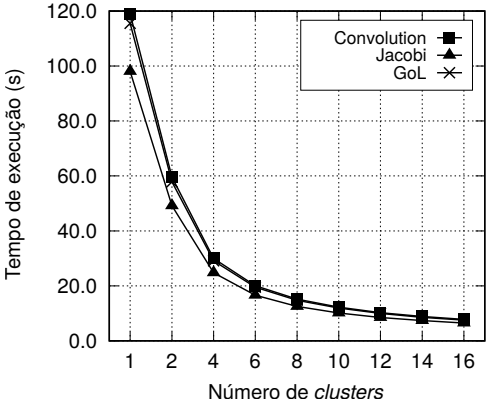
(a) Convolution (b) GoL (c) Jacobi

Fonte: o autor.

Outrossim, é observado que ao aumentar o tamanho do *tile* o desempenho das aplicações é muito beneficiado, independente do tamanho do dado de entrada. Isto está vinculado a duas circunstâncias. Primeiro, a redução do número de operações de escrita/leitura

e sincronizações entre o subsistema de E/S e os *clusters* por meio da NoC. Isso permite que maiores transferências de dados sejam realizadas por operação de escrita/leitura, otimizando a taxa de transferência da NoC. Segundo, maiores *tiles* significam maior paralelismo dentro dos *clusters* de computação (*i.e., threads* OpenMP possuirão mais trabalho para computar), reduzindo o *overhead* proveniente das regiões paralelas do OpenMP. Ao variar o tamanho do *tile* de 32x32 para 64x64, foram observados ganhos de até 3x em todas as aplicações. O ganho de desempenho aumentou em pelo menos 6,9x e 10,3x quando o tamanho do *tile* foi variado de 32x32 para 128x128 e de 32x32 para 256x256, respectivamente.

Figura 10 – Escalabilidade.



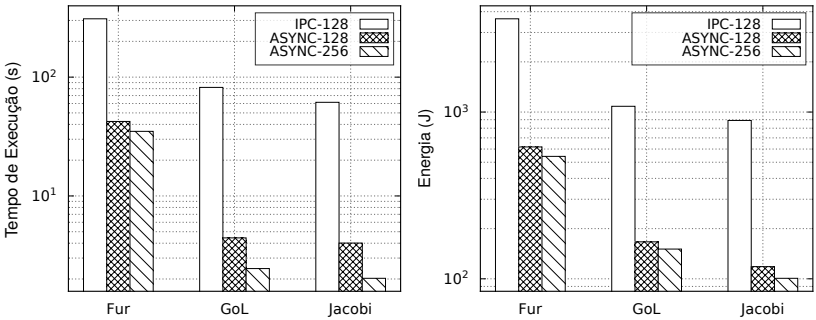
Fonte: o autor.

5.5 ANÁLISE DE ESCALABILIDADE

Nesta seção será analisada a escalabilidade do PSkel-MPPA na versão ASYNC. A Figura 10 mostra o tempo de execução de cada aplicação no MPPA-256, variando o número de *clusters* de computação utilizados de 1 a 16. Neste experimento, foi utilizado o número de iterações $t = 100$, o tamanho do dado de entrada e do *tile* de 12288x12288 e 256x256, respectivamente. Como pode ser visto, as três aplicações estêncil possuem comportamento semelhante e tem

o tempo de execução reduzido a medida que o número de *clusters* utilizados aumenta. Foi observada uma aceleração de 15,3x quando comparado o uso de 16 *clusters* em relação a apenas um *cluster*. Isso mostra que a versão PSkel-MPPA ASYNC consegue explorar o uso de todas as fontes de computação e o uso da NoC do MPPA-256.

Figura 11 – ASYNC *vs.* IPC.



Fonte: o autor.

5.6 MPPA-256 ASYNC *VS.* MPPA-256 IPC

Esta seção apresenta a comparação entre a nova versão do PSkel-MPPA (ASYNC) com a versão antiga do PSkel-MPPA (IPC). A Figura 11 apresenta a comparação de tempo de execução e consumo de energia entre ambas versões. Nesse experimento, foi utilizada uma matriz de dados de tamanho 12288x12288, *tiles* de tamanho 128x128 e 256x256 (apenas para a ASYNC), 16 *clusters* e 16 *Processing Elements* por *cluster*. A versão IPC recorria à alocação de dados temporários para auxiliar no envio/recebimento de dados dos *clusters* de computação, consumindo parcela significativa da já escassa memória local e limitando o tamanho máximo dos *tiles*. Esta prática foi descontiuada na versão ASYNC com o uso da nova API, viabilizando o aumento do tamanho máximo dos *tiles* para 256x256. O número de iterações utilizado foi $t = 30$. Os resultados obtidos representam a média de 5 execuções com o desvio padrão médio menor

que 1%, o número de iterações e execuções visa manter paridade com os resultados gerados na versão IPC (PODESTÁ JUNIOR, 2018).

O tempo de execução da versão ASYNC é de 8.86x, 33.48x e 30.29x mais rápido que o tempo de execução da versão IPC, para as aplicações Fur, GoL e Jacobi, respectivamente. Essa diferença está correlacionada ao fato da versão ASYNC fazer melhor uso da NoC na distribuição dos dados para os *cluster* de computação e realizar menos sincronizações entre mestre e trabalhador. Além disso, otimizações na computação como a verificação de elementos presentes nas bordas de computação também contribuem para este resultado.

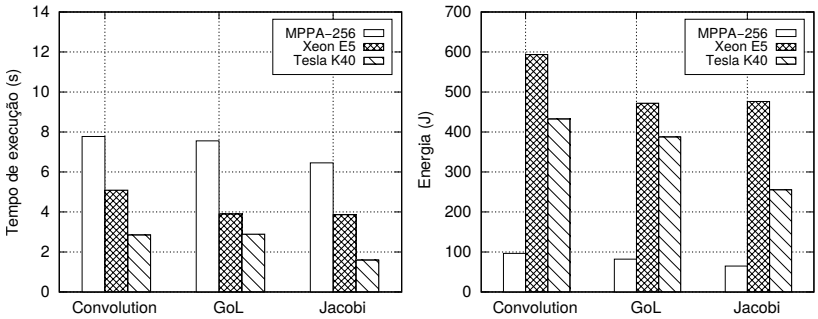
O consumo de energia segue um comportamento similar, uma vez que, menos tempo executando acarreta em menos tempo consumindo energia. Assim, é apresentada uma eficiência no consumo de energia superior em até 8.84x a favor da versão ASYNC.

5.7 MPPA-256 ASYNC *VS.* CPU *VS.* GPU

Por fim, é comparado o tempo de execução e consumo de energia obtido com o PSkel-MPPA ASYNC em relação as implementações do PSkel para CPU e GPU. Foram utilizados dados de entrada de tamanho 12288x12288 e *tiles* de tamanho 256x256, a escolha do tamanho dos *tiles* foi baseada no melhor desempenho obtido na Figura 9. Com o objetivo de realizar uma comparação justa, foram utilizadas as melhores otimizações disponíveis para o *Xeon E5* e *Tesla K40*, presentes nos *back-ends* de *multicore* e GPU do PSkel. O número de iterações utilizado foi $t = 100$, utilizando os 16 *clusters* e 16 *Processing Elements* por *cluster*. Os resultados obtidos representam a média de 20 execuções com o desvio padrão médio menor que 1%. A Figura 12 apresenta os resultados obtidos.

De modo geral, o PSkel-MPPA ASYNC é competitivo quanto ao tempo de execução, porém fica atrás das implementações para CPU e GPU. O tempo de execução das aplicações Convolution, GoL e Jacobi no MPPA-256 foi de 1.52x, 1.93x e 1.67x maior em relação a CPU e de 2.72x, 2.61x e 4.04x maior em relação a GPU, respectivamente.

Figura 12 – MPPA-256 ASYNC *vs.* CPU *vs.* GPU.



Fonte: o autor.

No consumo de energia o PSkel-MPPA ASYNC obteve os melhores resultados em todas as aplicações. O principal motivo é que o próprio MPPA-256 oferece uma arquitetura altamente paralelizável e com baixo consumo de energia. Ainda assim, a técnica de *tiling* trapezoidal utilizada (PODESTÁ JUNIOR, 2018), a implementação com a nova API de comunicação ASYNC e as otimizações realizadas foram de suma importância para obtenção desse desempenho no consumo energético. O consumo de energia no MPPA-256 foi de até 7.34x e 4.71x menor do que o da CPU e GPU, respectivamente.

6 CONTRIBUIÇÕES ACADÊMICAS

Este TCC teve a oportunidade de ser realizado paralelamente com uma bolsa de iniciação científica. Na etapa incial de desenvolvimento do projeto foi publicado um artigo (NASCIMENTO; JR.; CASTRO, Márcio, 2018) com os resultados obtidos para uma versão inicial, que já indicavam resultados promissores ao atuar na otimização da comunicação. Este, foi publicado na Escola Regional de Alto Desempenho 2018 (ERAD-RS 2018) e apresentado pelo autor na sessão de Arquiteturas + Aplicações em HPC do Fórum de Inicição Científica do evento.

Além disso, o trabalho final gerou resultados que mostraram-se significantes na área de processamento paralelo e distribuído. um artigo (PODESTÁ; NASCIMENTO; CASTRO, Márcio, 2018) foi aceito na maior conferência europeia de computação paralela e distribuida, a Euro-Par 2018 (International European Conference on Parallel and Distributed Computing 2018). Devido às restrições financeiras da universidade em auxiliar alunos da graduação em eventos internacionais, os alunos autores não puderam apresentar o artigo, o qual foi apresentado pelo orientador desta monografia.

7 CONCLUSÃO

É irrefutável a busca pelo ganho de desempenho atrelado ao aumento da eficiência energética. O futuro persegue incessantemente dispositivos de computação cada vez mais potentes e com menor consumo de energia. Neste trabalho essa busca se traduz na otimização de um *framework* desenvolvido para o processador MPPA-256 que é um *chip* de computação de alto desempenho voltado ao baixo consumo energético.

O desenvolvimento de aplicações otimizadas para processadores *manycore* de baixa potência 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. A nova API de comunicação assíncrona facilita o desenvolvimento do *back-end* do *framework* através do seu alto nível de abstração apresentado, contribuindo também para manutenabilidade do código fonte.

Neste trabalho foi apresentada uma nova versão otimizada do *framework* PSkel para o processador MPPA-256. Os resultados mostraram que a nova versão obteve ganhos de desempenho de até 33.48x e uma redução no consumo de energia de até 8.84x, em comparação com a solução inicial proposta em (PODESTÁ JUNIOR, 2018). Além disso, a solução mostrou-se competitiva quando comparada com outras plataformas de computação paralela, como *multicore* (CPU) e placas gráficas (GPU).

Como trabalhos futuros sugere-se a implementação para dar suporte a matrizes tridimensionais, assim como implementar técnicas de *prefetching* visando melhorar o desempenho do tempo de execução das aplicações. Além disso, sugere-se realizar comparações e análises com outras plataformas além da CPU e GPU como, por exemplo, outros processadores *manycore*.

REFERÊNCIAS

BUONO, D. et al. Parallel Patterns for General Purpose Many-Core. In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. [S.l.: s.n.], 2013. p. 131–139. DOI: 10.1109/PDP.2013.27. Citado 0 vez na página 43.

CASTRO, Marcio Bastos et al. Analysis of Computing and Energy Performance of Multicore, NUMA, and Manycore Platforms for an Irregular Application, 5:1–5:8, nov. 2013. Citado 1 vez nas páginas 23, 33.

CASTRO, Márcio et al. Seismic Wave Propagation Simulations on Low-power and Performance-centric Manycores. **Parallel Computing**, Elsevier, 2016. DOI: 10.1016/j.parco.2016.01.011. Disponível em: <https://hal.archives-ouvertes.fr/hal-01273153>. Citado 1 vez na página 23.

COLE, Murray. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. **Parallel Computing**, v. 30, n. 3, p. 389–406, 2004. ISSN 0167-8191. DOI: https://doi.org/10.1016/j.parco.2003.12.002. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167819104000080>. Citado 1 vez na página 23.

DEMMEL, James W. **Applied numerical linear algebra**. [S.l.]: SIAM, 1997. Citado 3 vezes nas páginas 36, 53.

FRANCESQUINI, Emilio et al. On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms. **Journal of Parallel and Distributed Computing**, Elsevier, v. 76, p. 32–48, fev. 2015. DOI: 10.1016/j.jpdc.2014.11.002. Disponível em: <https://hal-brgm.archives-ouvertes.fr/hal-01092325>. Citado 2 vezes nas páginas 23, 34, 43.

GARDNER, Martin. Mathematical Games - The Fantastic Combinations of John Conway's New Solitaire Game 'Life'. **Scientific American**, v. 223, 1970. ISSN 0036-8733. Citado 1 vez na página 53.

GORLATCH, Sergei; COLE, Murray. Parallel Skeletons. In: **Encyclopedia of Parallel Computing**. Edição: David Padua. Boston, MA: Springer US, 2011. p. 1417–1422. ISBN 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_24. Disponível em: <https://doi.org/10.1007/978-0-387-09766-4_24>. Citado 1 vez na página 35.

GYSI, T.; GROSSER, T.; HOEFLER, T. MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures. In: INTERNATIONAL Conference on Supercomputing (ICS). Irvine, USA: ACM, 2015. p. 177–186. Citado 0 vez na página 43.

HOLEWINSKI, Justin; POUCHET, Louis-Noël; SADAYAPPAN, P. High-performance Code Generation for Stencil Computations on GPU Architectures. In: PROCEEDINGS of the 26th ACM International Conference on Supercomputing. San Servolo Island, Venice, Italy: ACM, 2012. (ICS '12), p. 311–320. ISBN 978-1-4503-1316-2. DOI: 10.1145/2304576.2304619. Disponível em: <http://doi.acm.org/10.1145/2304576.2304619>. Citado 1 vez na página 23.

NASCIMENTO, Bruno Marques; JR., Emmanuel Podestá; CASTRO, Márcio. Otimização da Comunicação em Aplicações Estêncil Paralelas Implementadas com o PSkel no Processador MPPA-256. In: ANAIS da XVIII Escola Regional de Alto Desempenho do Rio Grande do Sul. 0: SBC, 2018. Disponível em: <https://sol.sbc.org.br/index.php/eradrs/article/view/2800>. Citado 1 vez na página 61.

PEREIRA, Alyson D.; RAMOS, Luiz; GÓES, Luís F. W. PSkel: A stencil programming framework for CPU-GPU systems. **Concurrency and Computation: Practice and Experience**, v. 27, n. 17, p. 4938–4953, 2015. cpe.3479. ISSN 1532-0634. DOI: 10.1002/cpe.3479. Disponível em: <http://dx.doi.org/10.1002/cpe.3479>. Citado 4 vezes nas páginas 24, 36, 53, 54.

PODESTÁ JR., Emmanuel et al. Execução Energeticamente Eficiente de Aplicações Estêncil com o Processador Manycore MPPA-256. In: XVIII Simpósio em Sistemas Computacionais de Alto Desempe-

nho (WSCAD). Campinas, SP: [s.n.], 2017. Disponível em: <https://portaldeconteudo.sbc.org.br/index.php/wscad/article/view/238>. Citado 3 vezes nas páginas 24, 25, 34, 36, 37.

PODESTÁ JUNIOR, Emmanuel. **PSkel-MPPA: Uma Adaptação do Framework PSkel para o Processador Manycore MPPA-256.** [S.l.: s.n.], 2018. Monografia (Bacharel em Cência da Computação), UFSC (Universidade Federal de Santa Catarina), Florianópolis, Brasil. Citado 9 vezes nas páginas 38–41, 44, 51, 53, 58, 59, 63.

PODESTÁ, Emmanuel; NASCIMENTO, Bruno Marques; CASTRO, Márcio. Energy Efficient Stencil Computations on the Low-Power Manycore MPPA-256 Processor. In: _____. **Euro-Par 2018: Parallel Processing**. Cham: Springer International Publishing, 2018. p. 642–655. ISBN 978-3-319-96983-1. Citado 1 vez na página 61.

ROCHA, Rodrigo C. O. et al. TOAST: Automatic tiling for iterative stencil computations on GPUs. **Concurrency and Computation: Practice and Experience**, v. 29, n. 8, p. 1–13, 2017. ISSN 1532-0634. DOI: 10.1002/cpe.4053. Citado 1 vez nas páginas 40, 41.

SOUZA, Matheus A. et al. CAP Bench: A Benchmark Suite for Performance and Energy Evaluation of Low-power Many-core Processors. **Concurrency and Computation: Practice and Experience**, 2016. ISSN 1532-0634. DOI: 10.1002/cpe.3892. Citado 0 vez na página 43.

TANENBAUM, Andrew S.; BOS, Herbert. **Modern Operating Systems**. 4th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. ISBN 013359162X, 9780133591620. Citado 6 vezes nas páginas 27–30.

THORARENSEN, S. et al. Efficient Execution of SkePU Skeleton Programs on the Low-Power Multicore Processor Myriad2. In: EUROMICRO International Conference on Parallel, Distributed, and Network-Based Processing (PDP). [S.l.: s.n.], 2016. p. 398–402. DOI: 10.1109/PDP.2016.123. Citado 0 vez na página 43.

TOTONI, Ehsan et al. 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). New Brunswick, Canada: IEEE Computer Society, 2012. p. 78–87. DOI: 10.1109/ISPASS.2012.6189208. Citado 0 vez na página 43.

WILENSKY, U. **NetLogo Fur model**. Center for Connected Learning e Computer-Based Modeling, Northwestern University, Evanston, IL: [s.n.], 2003. Disponível em: <http://ccl.northwestern.edu/netlogo/models/Fur>. Citado 1 vez na página 52.

Apêndices

APÊNDICE A – CÓDIGO FONTE

O código fonte deste trabalho está disponível em:
`https://github.com/b-marques/pskel/tree/mppaWidth`

APÊNDICE B – ARTIGO

Energy Efficient Stencil Computations on the Low-Power Manycore MPPA-256 Processor*

Emmanuel Podestá Jr., Bruno Marques do Nascimento, and Márcio Castro[0000−0002−9992−8540]

Graduate Program in Computer Science (PPGCC)
Federal University of Santa Catarina (UFSC)
Florianópolis, SC, Brazil
{emmanuel.podesta, bruno.mn}@grad.ufsc.br, marcio.castro@ufsc.br

Abstract. A new class of highly-parallel low-power manycore chips that cope with energy constraints have been unveiled. Sunway's SW26010 and Kalray's MPPA-256 are examples of them, featuring more than two hundred cores in a single low-power chip. Although they may present better energy efficiency than general-purpose multicore processors, architectural characteristics such as their limited amount of distributed on-chip memory make the development of efficient scientific parallel applications a challenging task. In this paper we propose and evaluate a new back-end of PSkel, a framework that provides a single high-level abstraction for stencil programming on CPUs and GPUs, for the low-power manycore MPPA-256 processor. This relieves programmers of the burden of explicitly dealing with communications and the hybrid underlying programming model of MPPA-256. Our results showed that the energy consumption of stencil applications running on MPPA-256 is up to 7.34x and 4.71x lower than on an Intel Xeon E5 multicore and NVIDIA Tesla K40 GPU, respectively.

Keywords: MPPA-256 · Manycore · PSkel · Energy efficiency.

1 Introduction

High Performance Computing (HPC) platforms have been evaluated based almost exclusively on their raw processing speed. However, their energy efficiency have become as important as raw performance. Because of that, a new class of highly-parallel low-power manycore chips that cope with energy constraints was unveiled. Sunway's SW26010 [6] and Kalray's MPPA-256 [5] are examples of such processors, providing more than two hundred low-power autonomous cores that can be exploited through both data and task parallelism.

* The authors would like to thank CAPES and CNPq for funding this research. This work was also supported by STIC-AmSud/CAPES scientific cooperation program under EnergySFE research project grant No. 99999.007556/2015-02. Finally, a special thank to Jean-François Méhaut (LIG/CNRS) for giving access to the MPPA-256 platform to the authors.

Although low-power manycores may present better energy efficiency than general-purpose multicore processors [5], their particular architectural characteristics make the development of efficient scientific parallel applications a very challenging task [2, 18]. Processing cores with non-coherent caches are usually distributed in a clustered architecture that features a hybrid programming model. On the one hand, cores in the same cluster share a limited amount of directly addressable memory. On the other hand, distinct clusters must communicate through the Network-on-Chip (NoC) in a distributed fashion. For that reason, communication costs between cores may vary significantly, depending on the location of the communicating cores on the NoC.

One possible approach to ease the development of parallel applications for low-power manycores is through the use of skeletons [3]. Skeletons allow programmers to focus on designing algorithms rather than worrying about synchronization issues and task scheduling, which are transparently handled by the skeleton framework, thereby speeding up application development and debugging. Among several existing patterns of parallel skeletons (*e.g.*, map, reduce, pipeline and scan), the stencil pattern has been used in applications of many important fields, such as quantum physics, weather forecasting and digital image processing [8]. The stencil pattern operates on n-dimensional data structures, using an input data value and its neighbors to compute the corresponding output data element. This process is repeated for every input data value in the n-dimensional data structure.

Indeed, many frameworks have been proposed to ease the development of parallel stencil computations on multicores and Graphics Processing Units (GPUs) such as PSkel [11], SkePU [16] and SkelCL [15]. In particular, PSkel is a stencil framework that provides a single high-level abstraction for stencil programming on heterogeneous CPU-GPU systems, while allowing automatic data partition, assignment and computation to both CPU and GPU. In this paper we present the design, implementation and evaluation of a new back-end of PSkel for the low-power manycore MPPA-256 processor (PSkel-MPPA). The same high-level, low overhead and intuitive PSkel code that already ran transparently on GPUs and multicores is extended to run also on the MPPA-256 architecture. This relieves programmers of the burden of explicitly dealing with NoC communications, the hybrid underlying programming model and the absence of cache coherence on MPPA-256. Our solution uses a trapezoidal tiling technique to reduce the number of communications and synchronization barriers on MPPA-256, which improves considerably the overall performance. Our results show that the energy consumption of stencil applications on MPPA-256 is up to 7.34x and 4.71x lower than on an Intel Xeon E5 multicore and NVIDIA Tesla K40 GPU, respectively, while presenting competitive performance.

The remainder of this paper is organized as follows. Section 2 presents an overview of MPPA-256 and PSkel. Next, Section 3 describes our proposal (PSkel-MPPA) as well as its implementation details. Then, Section 4 presents the results obtained with PSkel-MPPA, comparing them against reference implementations

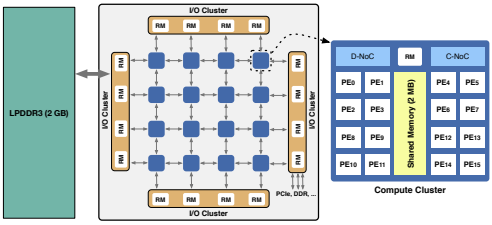


Fig. 1. Overview of the low-power MPPA-256 manycore processor.

of PSkel for multicores and GPUs. Section 5 discusses related work. Finally, Section 6 concludes this paper.

2 Background

2.1 MPPA-256

MPPA-256 is a single-chip low-power manycore processor developed by Kalray that integrates 256 user cores and 32 system cores in 28 nm CMOS technology running at 400 MHz. These cores are distributed across 16 compute clusters and 4 Input/Output (I/O) clusters that communicate through data and control NoCs. The board used in this paper has one of the I/O clusters connected to an external Low-Power Double Data Rate 3 (LPDDR3) of 2 GB. Fig. 1 shows an architectural overview of the MPPA-256 processor.

Overall, each compute cluster has the following components:

- 16 cores called Processing Elements (PEs), which are dedicated to run user threads (one thread per PE) in non-interruptible and non-preemptible mode. Each PE has private 2-way associative 32 kB instruction and data caches;
- One Resource Manager (RM), which is responsible for running the operating system and managing communications;
- A low-latency shared memory of 2 MB, which enables a high bandwidth and throughput between PEs within the same compute cluster; and
- Two NoC controllers, one for data and other for control.

The processor features a distributed memory model. Compute clusters and the I/O clusters have their own address spaces. Applications must use two parallel programming libraries to exploit all processor resources: a thread library (Pthread or OpenMP) and a proprietary library called Asynchronous Operations Application Programming Interface (Async API). The former is used to parallelize computations in computing clusters via shared memory. The latter follows a distributed memory model and must be used for cluster-cluster and cluster-I/O communications through the NoC.

Async API is based on one-sided communications between the compute clusters' local memory and LPDDR3. The main concepts behind Async API are *execution domains*, *segments* and `put`/`get` operations. The execution domain represents a set of cores sharing a local memory, being isolated from other execution domains. Considering the MPPA-256 distributed memory model, an execution domain corresponds to a compute cluster or to an I/O cluster. Memory that is not directly accessible from the cores of an execution domain can be structured into segments, which correspond to the entire or part of the local memory of cores located in another execution domain. Each segment has a *unique signature*, which is specified when the segment is created in an execution domain through the `mppa_async_segment_create()` function. Then, other execution domains can reference a previously created segment by passing its unique signature the function `mppa_async_segment_clone()`. Once segments are created and referenced by different execution domains, one should use put/get operations to read data from a remote segment into the local memory (`get` operation) or to write local data to the remote segment (`put` operation). Different flavors of these operations are available in Async API, allowing contiguous or spaced data transfers (*e.g.*, `mppa_async_put()` and `mppa_async_get_spaced()`) as well as 2D block transfers (`mppa_async_sget_block2d()`), which is useful for transferring 2D data blocks.

The execution flow of an MPPA-256 application is the following. The main process (called *master process*) runs on an RM of the I/O cluster connected to the LPDDR3 and is responsible for allocating the input data in its local memory (LPDDR3) and spawning *worker processes* (one for each compute cluster) by calling the `mppa_power_base_spawn()` function. The necessary data segments should be created by the master process so it can exchange data with the compute clusters. Finally, the master process should wait all worker processes to finish by calling the `mppa_power_base_waitpid()` function. Each worker process should make references to remote segments allocated in the LPDDR3 to exchange data during the execution and may create up to 16 threads using Pthread or OpenMP (one thread for each PE) to perform computations in parallel. Each PE has its own private cache memory without any automatic coherence mechanism among the remaining PEs cache memories. Although this improves the cache performance, it requires the developer to explicitly flush data when needed.

2.2 Stencil Pattern and PSkel

The stencil computational pattern operates on n-dimensional data structures and uses a sliding window (*a.k.a* mask) that scans the entire input data set and produces output data using a user-defined stencil kernel function. The mask size corresponds to a specific number of neighbors of each element of the input data. The stencil application repeats that process on every element of the input data. Stencil applications can be iterative, which means that the output data produced after an iteration t is used as the input for an iteration $t + 1$.

PSkel is a framework for high-level programming stencil computations, based on the concept of parallel skeletons, which offers parallel execution support on

```
1  __parallel__ void stencilKernel(Array2D<float> A, Array2D<float> B,
2                                  struct Arguments args, int x, int y) {
3    B(x,y) = args.alpha*(A(x,y+1)+A(x,y-1)+A(x+1,y)+A(x-1,y)+args.beta);
4  }
5
6  void jacobi(float *A, float *B, int M, int N, float alpha, float beta,
7              int timesteps) {
8    Array2D<float> input(A,M,N);
9    Array2D<float> output(B,M,N);
10   struct Arguments args(alpha, beta);
11   Stencil2D<Array2D<float>, struct Arguments> stencil(input,output,args);
12   stencil.runIterativeGPU(timesteps);
13 }
```

Fig. 2. Simplified example of a PSkel stencil code.

CPUs and GPUs [11]. PSkel offers a single programming interface, decoupled from the runtime back-ends, that releases the programmer from the responsibility of writing boiler-plate code for parallel stencil computation. Instead, the programmer is responsible for implementing a stencil kernel describing solely the computation, while the framework translates the abstractions described into low-level parallel C++ code. Synchronization, memory management and data transfers are transparently handled by the framework.

Fig. 2 shows an example of the Jacobi method for solving matrix equations [4] written in PSkel. The PSkel Application Programming Interface (API) provides templates for manipulating input and output data via template classes for n-dimensional arrays, called `Array`, `Array2D` (Fig. 2, lines 8–9) and `Array3D`. These abstractions provide methods that encapsulate the data management procedures, such as memory allocation, memory copy and data transfer (*e.g.*, communication between CPU and GPU). Moreover, it provides abstractions for specifying the stencil kernel and to manage the stencil execution. The stencil kernel (prototype function `stencilKernel()`) is the application specific method that describes the computation that will be performed on each entry of the input array and its neighbors (Fig. 2, lines 1–4). The `stencilKernel()` prototype function must be implemented by the user of the PSkel. Finally, the API provides a set of classes for managing the whole execution of the user-defined number of iterations of the stencil kernel over the input and output data, such as `Stencil`, `Stencil2D` (Fig. 2, line 11) and `Stencil3D`.

In the given example, the `stencilKernel()` function will be executed on the GPU. The `runIterativeGPU()` method hides from the user all the CUDA code needed to correctly execute the specified stencil kernel on the GPU.

3 PSkel-MPPA

As previously discussed in Section 2.2, PSkel currently supports the execution of stencil applications on CPUs and GPUs. In this paper we propose a new back-end of PSkel for the low-power manycore MPPA-256 processor, which differs significantly from the CPU and GPU ones due to the intrinsic characteristics

of MPPA-256 discussed in Section 2.1, such as: (i) limited amount of on-chip memory; (ii) clustered architecture with NoC constraints; (iii) processing cores with non-coherent caches; and (iv) proprietary low-level communication API.

The new back-end, named PSkel-MPPA, supports 2D stencils (`Stencil2D` class in PSkel) and adopts the master-worker model. The master process is executed in the I/O cluster connected to the LPDDR3 memory, in which the input and output data (`Array2D` objects) are allocated, whereas the worker processes are executed on the compute clusters (one worker process per compute cluster) to perform the stencil computation in parallel. Given the memory limitation inside compute clusters (2 MB), the input `Array2D` is partitioned into *tiles* of fixed user-defined size to be sent to them. When tiling stencil computations, neighborhood dependencies inherent to the stencil parallel pattern must be considered before partitioning the input data.

We used the trapezoidal tiling technique to handle neighborhood dependencies in PSkel-MPPA, resulting in redundant data and computation per tile [13]. We use a formal definition to illustrate this technique. Let A be a 2D data matrix, with dimensions $\dim(A) = (w, h)$, where w and h are, respectively, its width and height. Using tiles of dimensions (w', h') yields $\lceil \frac{w}{w'} \rceil \lceil \frac{h}{h'} \rceil$ possible tiles of A. Let $A_{i,j}$ be one such tile, where $0 \leq i < \lceil \frac{w}{w'} \rceil$ and $0 \leq j < \lceil \frac{h}{h'} \rceil$. $A_{i,j}$ has offset (iw', jh') relative to the top left corner of A and $\dim(A_{i,j}) = (\min\{w', w - iw'\}, \min\{h', h - jh'\})$. The offset is an indexing displacement required for accessing the elements of the tile.

Fig. 3 shows a graphical view of this technique. A logical tile (inner solid line) is contained in a 2D data matrix (outer dashed line) with vertical and horizontal offsets given by jh' and iw'. If t iterations of a stencil application should be executed, it is possible to compute t' consecutive iterations on $A_{i,j}$ ($t' \in [1, t]$) without the need of any data exchange between adjacent tiles (*a.k.a* inner iterations). To do so, the logical tile ($A_{i,j}$) must be enlarged with a ghost zone (area between the inner solid line and the outer solid line), which is comprised of a halo region (the area between the inner solid line and the inner dashed line). Let r be the most distant displacement required for the neighborhood defined by the stencil mask. The area of range r comprising the neighborhood is denominated *halo region*. The number of adjacent halo regions that compose the ghost zone is proportional to t'. Thus, the enlarged tile $A^*_{i,j}$ has offsets $(\max\{iw' - rt', 0\}, \max\{jh' - rt', 0\})$ relative to A. Thus, sizing the ghost zones poses a trade-off between the cost of redundant computations and the reduction in communication and synchronizations on the NoC when processing iterative stencil computations on MPPA-256.

The execution flow of PSkel-MPPA follows the one described in Section 2.1. During the initialization phase, the master process running on the I/O cluster allocates the input and output data in the LPDDR3, and creates a specific segment for each one of them. Next, it calculates the number of enlarged tiles that will be produced as well as their dimensions based on: i) user-defined parameters, such as the input data and logical tile dimensions, the number of compute clusters and the number of inner iterations; and ii) stencil kernel parameters, such

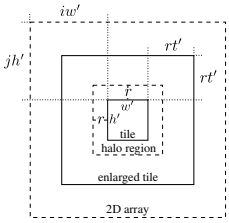


Fig. 3. 2D tiling [13].

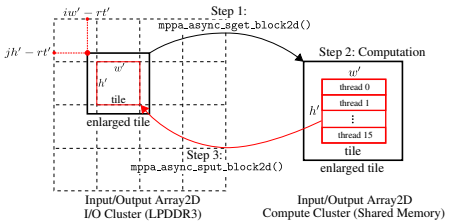


Fig. 4. Communications with `block2d`.

as the mask size. Then, it spawns up to 16 worker processes (one for each compute cluster) and informs each worker process about the number of enlarged tiles produced, their dimensions and the subset of tiles it should compute later on. Finally, the master process waits for all workers to finish. Each worker process, on the other hand, allocates data to store the input and output enlarged tiles in the compute cluster local memory and clones both input and output remote segments that were already created by the master process to make data transfers further on. The initialization phase in both master and worker processes is encapsulated in the `Stencil2D` class.

The computation phase consists of the execution of the stencil kernel by the worker processes. The following three main steps are performed to compute each tile assigned to a worker process: 1) the enlarged tile is extracted from the input data allocated in LPDDR3 and transferred to compute cluster local memory to be processed; 2) t' iterations of the stencil kernel (inner iterations) are executed by the worker process over the enlarged tile; and 3) the resulting logical tile is transferred back from the compute cluster local memory to its corresponding position in the LPDDR3. Once all tiles assigned to each worker process were successfully computed, all worker processes must synchronize at a global barrier, since the data computed during t' iterations will be needed by the others in the following iteration to solve neighborhood dependencies. We used the `mppa_rpc_barrier_all()` function for this purpose. The whole procedure described before is then repeated until the total number of iterations defined by the user (t) is reached.

The aforementioned steps are depicted in Fig. 4 and they are described in more detail below:

Step 1. Based on the information given by the master process during the spawn procedure, the worker process is capable of calculating the coordinates of each enlarged tile assigned to it with respect to the input data allocated in the LPDDR3 ($iw' - rt'$ and $jh' - rt'$ coordinates) without any other intervention from the master process. The `mppa_async_sget_block2d()` function takes such information and the block size as input parameters and it transfers the

enlarged tile to be processed by the worker process from the input remote segment into the compute cluster local memory through the NoC.

Step 2. The worker process computes t' iterations of the user-defined stencil kernel over the enlarged tile. In each t' iteration, the computation is parallelized by means of an OpenMP parallel region. The parallel region creates up to 16 threads (one for each PE). Each PE is responsible for executing the stencil kernel on a subset of the enlarged tile cells.

Step 3. After the stencil kernel computation, the resulting logical tile is transferred back to the LPDDR3. The `mppa_async_sput_block2d()` function is used for this purpose, allowing the logical tile to be extracted from the enlarged tile in the compute cluster local memory and transferred to its corresponding position in the output remote segment.

Fortunately, all complex tasks related to the tiling technique, NoC communications and adaptations discussed in this section are hidden from the developers, since they were included in the back-end of PSkel. This means that current applications developed with the PSkel framework can run seamlessly on MPPA-256 without any source code modifications.

4 Experimental Evaluation

4.1 Platforms, Applications and Inputs

We evaluate the performance and energy consumption of the proposed solution (PSkel-MPPA) against reference multicore and GPU implementations available in PSkel. Energy measurements were collected from power and energy sensors available on MPPA-256, which include all clusters, memory (on-chip memory and LPDDR3) and NoCs. The reference implementations of PSkel for CPUs and GPUs were executed on the platforms described bellow. Compilation was done using GCC 5.4 (MPPA-256 and CPU) and NVCC version 8.0 (GPU) with the flags `-O3` (all platforms), `-march=native -mtune=native -ftree-vectorize` (CPU and GPU) and `-arch=sm_35` (GPU).

- **Xeon E5**: a desktop server featuring an Intel Xeon E5-2640 v4 (Broadwell) processor with 10 physical cores running at 2.4 GHz and 64 GB of RAM. Energy measurements on this platform are based on Intel's Running Average Power Limit (RAPL) interface, which considers the power consumption of hardware components through hardware counters. We used this approach to obtain the energy consumption of the CPU (`PACKAGE_ENERGY`) and DRAM (`DRAM_ENERGY`).
- **Tesla K40**: a NVIDIA Tesla K40c graphics board featuring 2880 CUDA parallel-processing cores with a base clock of 745 MHz and 12 GB of GDDR5 GPU memory. Energy measurements on this platform were obtained from NVIDIA Management Library (NVML). We used the NVML to gather the power usage for the GPU and its associated circuitry (*e.g.*, internal memory).

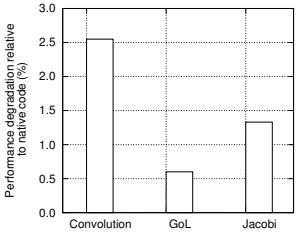


Fig. 5. Performance degradation of PSkel-MPPA (in percentage) relative to hand-optimized code for MPPA-256.

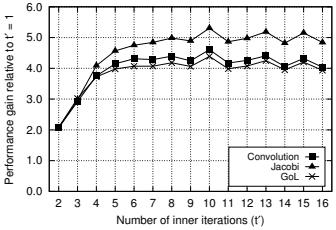


Fig. 6. Empirical study to find the best value for t'. The best trade-off is achieved with $t' = 10$.

We carried out several experiments with three stencil applications[1] implemented in PSkel: i) Convolution, which implements a classical convolution method used in signal and image processing; ii) GoL, which is a cellular automaton implementing Conway's Game of Life; and iii) Jacobi, which is an iterative method for solving matrix equations [4]. We also considered four input data sizes (2048x2048, 4096x4096, 8192x8192 and 12288x12288) to evaluate the performance of the aforementioned PSkel applications on MPPA-256, Xeon E5 and Tesla K40. Moreover, we evaluated the performance impacts of using different tile sizes (32x32, 64x64, 128x128 and 256x256) on MPPA-256, since input/output data sizes do not fit into compute clusters memory (2 MB). The maximum input/output and tile sizes were chosen carefully to fill MPPA-256 memories (2 GB of LPDDR3 and 2 MB of local memory in compute clusters). Finally, we fixed the number of iterations for each application to $t = 100$ in all experiments. All results represent averages of 20 runs with a maximum standard deviation of less than 1%.

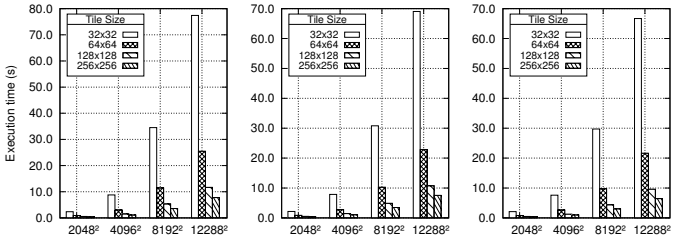
4.2 Overhead of PSkel

We first analyze the overhead introduced by our new back-end of PSkel (PSkel-MPPA). Fig. 5 shows the performance degradation of all three stencil applications implemented with PSkel-MPPA compared to hand-optimized ones implemented without PSkel abstractions. As it can be observed, the performance degradation introduced by PSkel-MPPA is minimal when compared to MPPA-256 native stencil code (less then 2.6%).

4.3 Sizing the Ghost Zone

In our solution, the trapezoidal tiling technique allows us to easily fine tune the size of the ghost zone with the t' parameter. Indeed, this is an important feature

[1] A detailed description about these PSkel applications is not presented in this paper due to space constraints but can be found in [11, 12].



(a) Convolution. (b) GoL. (c) Jacobi.

Fig. 7. Impact of tile size on the performance of the stencil applications.

to exploit the low amount of on-chip memory and to make a better use of the NoC available in low-power manycores. Thus, we carried out an empirical study with the aforementioned applications to determine the best value for this parameter. Fig. 6 shows the performance gains when varying t' from 2 to 16 (performance gains are relative to $t' = 1$). As we mentioned earlier, sizing the ghost zone poses a trade-off between the cost of redundant computations and the reduction in communication and synchronizations on the NoC. Our empirical study shows that the best trade-off is achieved when $t' = 10$ (performance obtained with $t' > 16$ varied around 4 and were omitted from the figure). Because of that, all results presented in next sections were carried out with $t' = 10$.

4.4 Tile Size vs. Performance

In this section, we analyze the impact of the tile size on the performance of PSkel applications on MPPA-256. Fig. 7 shows the performance of the stencil applications when varying the input data size and the tile size. Overall, we observed an average increase in the execution time of the applications between 2x and 3.3x as we double the input size. This behavior is expected since more communications and synchronizations must be performed for larger data inputs.

Moreover, we observed that the performance of the applications is greatly improved as we increase the tile size, regardless of the input size. The main reason for that is twofold. On the one hand, the number of `put`/`get` operations and synchronizations between the I/O and compute clusters on the NoC is greatly reduced as we increase the tile size. This allows for bigger data transfers per `put`/`get` operation, improving the NoC throughput. On the other hand, bigger tiles mean higher parallelism inside compute clusters (*i.e.*, OpenMP threads will have more work to compute), reducing the overhead imposed by OpenMP parallel regions. When varying the tile size from 32x32 to 64x64, we observed improvements of up to 3x on all applications. The performance gains increase to at least 6.9x and 10.3x when varying the tile size from 32x32 to 128x128 and from 32x32 to 256x256, respectively.

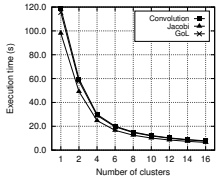


Fig. 8. Scalability.

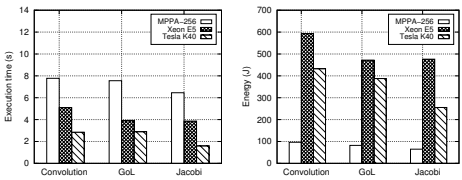


Fig. 9. Performance and energy comparison.

4.5 Scalability Analysis

Next, we analyze the scalability of PSkel-MPPA. Fig. 8 shows the execution time of each application on MPPA-256 when varying the number of compute clusters from 1 to 16. For this experiment, we used input data and tiles of size 12288x12288 and 256x256, respectively. As it can be noticed, all stencil applications present a similar behavior and have their execution times reduced as we increase the number of compute clusters. Overall, we observed a speedup gain of 15.3x with 16 compute clusters over the execution with a single compute cluster. This means that our solution is able to exploit all computing resources and the NoC of MPPA-256.

4.6 Comparison with CPU and GPU: Performance vs. Energy

Finally, we compare the execution time and energy consumption achieved by PSkel-MPPA against reference implementations of PSkel for CPU and GPU. In these experiments, we used input data of size 12288x12288. Based on the best performance achieved on Fig. 7, we used tiles of size 256x256 on MPPA-256. To make a fair comparison, we used the best tiling optimizations for Xeon E5 and Tesla K40 that were available in the multicore and GPU back-ends of PSkel, respectively. Fig. 9 presents the results obtained with all stencil applications.

Overall, PSkel-MPPA achieves competitive execution times compared to the CPU and GPU counterparts. As expected, the best performance was achieved on the GPU, since it has much more processing power than the other processors. The execution times of Convolution, GoL and Jacobi on MPPA-256 were 1.52x, 1.93x and 1.67x higher than on CPU, respectively. On the other hand, the execution times of Convolution, GoL and Jacobi on MPPA-256 were 2.72x, 2.61x and 4.04x higher than on GPU, respectively.

PSkel-MPPA achieved the best results with respect to the energy consumption on all applications. The main reason is that MPPA-256 offers a high parallelism and yet has a low power consumption. As we showed in Section 4.3, the trapezoidal tiling technique implemented in PSkel-MPPA was extremely important to achieve such energy improvements. We observed that the energy consumption on MPPA-256 was up to 7.34x and 4.71x lower than on the CPU and GPU, respectively.

5 Related Work

Due to the importance of parallel skeletons, and specifically the stencil parallel pattern, many recent efforts in research sought to improve the performance and broaden the support of skeletons on manycore processors. *Buono et al.* [1] ported a framework based on parallel skeletons, called FastFlow, to the manycore processor TilePro64. The TilePro64 has 64 identical processing cores interconnected by a mesh of network-on-chip. Similarly, *Thorarensen et al.* [16] presented a new back-end of the SkePU framework for the low-power manycore Myriad2. It features a heterogeneous architecture, targeting power constrained devices and mainly computer vision applications. *Lutz et al.* [9] used tiling techniques in stencil computations on multi-GPU environments by using the GPU memories collectively. Similarly, *Gysi et al.* [7] propose a framework for automatic tiling optimizations of stencil computations on CPU-GPU hybrid systems.

Recent works studied the performance and/or the energy efficiency of low-power manycore processors. *Totoni et al.* [17] compared the power and performance of Intel's Single-Chip Cloud Computer (SCC) to other types of CPUs and GPUs. Although they showed that there is no single solution that always achieves the best trade-off between power and performance, the results suggest that manycores are an opportunity for the future. *Morari et al.* [10] proposed an optimized implementation of radix sort for the Tilera TILEPro64 manycore processor. The results showed that their solution for TILEPro64 provides much better energy efficiency than an general-purpose multicore processor (Intel Xeon W5590) and comparable energy efficiency with respect to a GPU NVIDIA Tesla C2070. *Souza et al.* [14] proposed a benchmark suite to evaluate MPPA-256 manycore processor. The benchmark offers diverse applications regarding parallel patterns, job types, communication intensity and task load strategies, suitable for a broad understanding of performance and energy consumption of MPPA-256 and upcoming manycores. *Francesquini et al.* [5] evaluated three different classes of applications (CPU-bound, memory-bound and mixed) using highly-parallel platforms such as MPPA-256 and a 24-node, 192-core NUMA platform. They showed that manycore architectures can be very competitive, even if the application is irregular in nature. Using the Adapteva's Epiphany-IV low-power manycore, *Varghese et al.* [18] described how a stencil-based solution to the anisotropic heat equation using a two-dimensional grid was developed. This manycore has a low power budged (2 W) and has 64 processing cores. Similar to MPPA-256, Epiphany-IV has a very limited amount of local memory available to each core and no automatic prefetching mechanism exists; every data movement has to be explicitly controlled by the application.

To the best of our knowledge, PSkel-MPPA is the first complete implementation of a parallel stencil framework on MPPA-256. Our solution relieves programmers of the burden of explicitly dealing with NoC communications, the hybrid underlying programming model and the absence of cache coherence on MPPA-256. The trapezoidal tiling technique allows developers to fine tune the trade-off between the cost of redundant computations and the reduction in com-

munication and synchronizations on the NoC when processing iterative stencil computations on MPPA-256.

6 Conclusion

Low-power manycores have emerged as a building block for constructing energy-efficient HPC platforms. However, the development of efficient parallel applications is very challenging on these processors because developers must deal with hybrid programming models, limited amount of directly addressable memory and NoC constraints. In this paper, we propose to ease the development of stencil applications on the low-power MPPA-256 manycore processor by means of parallel skeletons. More precisely, we proposed a new back-end of the PSkel stencil framework for MPPA-256 named PSkel-MPPA, providing a single high-level abstraction for stencil programming on CPUs, GPUs and MPPA-256. Our solution relieves programmers of the burden of explicitly dealing with communications and the hybrid underlying programming model of MPPA-256.

The trapezoidal tiling technique adopted in our solution was essencial to exploit the low-power MPPA-256 manycore processor, improving the performance of our solution. Our results showed that PSkel-MPPA achieved the best results with respect to the energy consumption on all applications, being up to 7.34x and 4.71x more energy efficient than on the CPU and GPU considered in this study, respectively. Moreover, PSkel-MPPA achieved competitive performance on MPPA-256 in comparison to the CPU and GPU reference implementations. The GPU achieved the best performance, since it has much more processing power than the other processors.

As future works, we intend to extend our support in PSkel-MPPA for 3D stencils. In this case, it would be necessary to consider a prefetching scheme to overlap communications with computations. Moreover, we intend to compare our results on MPPA-256 against low-power ARM processors, which may also include a low-power GPU. Finally, we intend to provide similar abstractions for dealing with other kinds of skeletons.

References

1. Buono, D., Danelutto, M., Lametti, S., Torquati, M.: Parallel patterns for general purpose many-core. In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp. 131–139 (2013). https://doi.org/10.1109/PDP.2013.27
2. Castro, M., Francesquini, E., Dupros, F., Aochi, H., Navaux, P.O., Méhaut, J.F.: Seismic wave propagation simulations on low-power and performance-centric manycores. Parallel Computing **54**, 108–120 (2016). https://doi.org/10.1016/j.parco.2016.01.011
3. Cole, M.: Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. Parallel Comput. **30**(3), 389–406 (Mar 2004)
4. Demmel, J.W.: Applied numerical linear algebra. SIAM (1997)

5. Francesquini, E., Castro, M., Penna, P.H., Dupros, F., de Freitas, H.C., Navaux, P.O.A., Méhaut, J.F.: On the energy efficiency and performance of irregular applications on multicore, NUMA and manycore platforms. J. Parallel Distrib. Comput. **76**, 32–48 (2014). https://doi.org/10.1016/j.jpdc.2014.11.002
6. Fu, H., et al.: The sunway taihulight supercomputer: System and applications. SCIENCE CHINA Information Sciences **59**(7), 1–16 (2016). https://doi.org/10.1007/s11432-016-5588-7
7. Gysi, T., Grosser, T., Hoefler, T.: MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In: International Conference on Supercomputing (ICS). pp. 177–186. ACM, Irvine, USA (2015)
8. Holewinski, J., Pouchet, L.N., Sadayappan, P.: High-performance code generation for stencil computations on GPU architectures. In: International Conference on Supercomputing (ICS). pp. 311–320. ACM, Venice, Italy (2012)
9. Lutz, T., Fensch, C., Cole, M.: PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems. ACM Trans. Archit. Code Optim. **9**(4), 59:1–59:24 (2013)
10. Morari, A., Tumeo, A., Villa, O., Secchi, S., Valero, M.: Efficient sorting on the Tilera manycore architecture. In: International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). pp. 171–178. IEEE Computer Society, New York, USA (2012)
11. Pereira, A.D., Ramos, L., Góes, L.F.W.: PSkel: A stencil programming framework for cpu-gpu systems. Concurrency and Computation: Practice and Experience **27**(17), 4938–4953 (2015)
12. Pereira, A.D., Rocha, R.C.O., Castro, M., Goes, L.F.W., Dantas, M.A.R.: Extending OpenACC for efficient stencil code generation and execution by skeleton frameworks. In: International Conference on High Performance Computing & Simulation (HPCS). pp. 719–726. IEEE Computer Society, Genoa (2017). https://doi.org/10.1109/HPCS.2017.110
13. Rocha, R.C.O., Pereira, A.D., Ramos, L., Ges, L.F.W.: TOAST: Automatic tiling for iterative stencil computations on GPUs. Concurrency and Computation: Practice and Experience **29**(8), 1–13 (2017). https://doi.org/10.1002/cpe.4053
14. Souza, M.A., et al.: CAP bench: A benchmark suite for performance and energy evaluation of low-power many-core processors. Concurrency and Computation: Practice and Experience (2016). https://doi.org/10.1002/cpe.3892
15. Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In: IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW). pp. 1176–1182. IEEE Computer Society, Shanghai, China (2011)
16. Thorarensen, S., Cuello, R., Kessler, C., Li, L., Barry, B.: Efficient execution of skepu skeleton programs on the low-power multicore processor myriad2. In: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP). pp. 398–402 (2016). https://doi.org/10.1109/PDP.2016.123
17. Totoni, E., Behzad, B., Ghike, S., Torrellas, J.: 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). pp. 78–87. IEEE Computer Society, New Brunswick, Canada (2012). https://doi.org/10.1109/ISPASS.2012.6189208
18. Varghese, A., Edwards, B., Mitra, G., Rendell, A.P.: Programming the Adapteva Epiphany 64-core network-on-chip coprocessor. In: International Parallel Distributed Processing Symposium Workshops (IPDPSW). pp. 984–992. IEEE Computer Society, Phoenix, USA (2014)