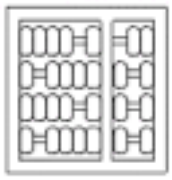


INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Construindo um Runtime OpenMP 4.0 com suporte a offload de resolução de dependências para acelerador em FPGA

Lucas Henrique Morais
Prof. Guido Costa de Souza Araújo

Relatório Técnico - IC-PFG-16-19 - Projeto Final de Graduação

December - 2016 - Dezembro

Construindo um Runtime OpenMP 4.0 com suporte a offload de resolução de dependências para acelerador em FPGA

Lucas Henrique Morais, Guido Costa de Souza Araújo

Resumo. As barreiras tecnológicas para a implementação de processadores unicore que mantivessem o ritmo de avanço de performance das últimas décadas deram ensejo para o surgimento das arquiteturas multicore que os desenvolvedores de hoje devem aprender a utilizar. A natural dificuldade de se desenvolverem sistemas destinados a tais arquiteturas levaram ao surgimento de diversos frameworks de programação paralela que procuram, cada um a seu modo, tornar tal tipo de desenvolvimento mais eficiente e menos propenso a erros. Nesse contexto surge o conceito de Task Parallelism, capaz de encapsular chamadas de função e pequenos trechos de código em entidades chamadas Tasks, que, sendo um análogo macroscópico das intruções de processador, podem ser escalonadas em paralelo nas diversas unidades de processamento disponíveis desde que suas relações de dependência - inferidas a partir do modo como cada uma acessa os seus dados - sejam respeitadas, numa lógica muito próxima à que rege o escalonamento de instruções para as diversas ALUs disponíveis em um processador superescalar. Atualmente, a especificação OpenMP dá amplo suporte a Task Parallelism, o que criou ocasião para o presente trabalho: o desenvolvimento de um runtime com suporte a OpenMP 4.0 capaz de acelerar task-scheduling por meio de offload de computação relativa à inferência de dependências de dados para um acelerador em FPGA.

Palavras-Chave: Task Parallelism; Computation offloading; OpenMP; Parallel computing; ARM (Computer architecture).

Introdução

O paradigma de programação paralela baseado em tasks (*Task Parallelism*) apresenta-se como um meio altamente eficiente, considerando-se o tempo de programação exigido para tal, de se paralelizarem bases de código seriais. Ao mesmo tempo, embora esse paradigma seja genérico o suficiente para poder modelar a execução paralela de uma ampla gama de aplicações, o desempenho de programas que o empreguem depende fortemente das características do runtime em software utilizado, que é o agente responsável por prover as abstrações definidas pelo paradigma. Sendo assim, o presente trabalho dedicou-se a propor um runtime em software de Task Parallelism que, utilizando-se de offload de computação para um acelerador dedicado em FPGA, pudesse apresentar desempenho superior ao dos runtimes sem semelhante recurso.

Justificativa

Enquanto a programação paralela baseada em primitivas básicas – como as de pthreads, por exemplo – dá liberdade suficiente ao programador para que cenários de erros difíceis de depurar surjam com certa frequência, o modelo de *Task Parallelism* demanda do programador que ele apenas selecione os trechos mais importantes do código por paralelizar, define esses segmentos como *tasks* e indique corretamente o modo como essas *tasks* acessam os dados lidos e/ou escritos por elas. Feito isso, tornam-se responsabilidade exclusiva do runtime de *Task Parallelism* (1) inferir as relações de dependência entre as tasks e (2) as escalonar aos processadores disponíveis de um modo que se respeitem essas dependências.

Sendo assim, já que o código paralelizado com *tasks* difere da sua versão serial, em geral, apenas pela inclusão que algumas anotações de código para indicar quais segmentos de código e descrever seus correspondentes modos de acesso aos seus dados, códigos *Task Parallel* costumam ser muito mais legíveis do que versões paralelizadas com *frameworks* alternativos, o que facilita sua manutenção.

Além disso, embora haja hoje uma ampla gama de ambientes de programação paralela (como TBB, GCD, OpenMP com seus diversos paradigmas), as abstrações de alto nível proporcionadas por essas ferramentas costumam ser mais difíceis de empregar do que a abstração básica de *tasks*, demandando, por vezes, um considerável esforço de *redesign* do código alvo – embora até possam, em alguns casos, levar a um desempenho superior que o dado pelo atual estado dos ambientes de *Task Parallelism*.

Dito isso, há razão suficiente para crer que, caso as ferramentas de suporte a *Task Parallelism* – *runtime systems*, *back-end* de compilação, etc – sejam capazes de extrair das

aplicações assim paralelizadas um desempenho competitivo ao oferecido por outras técnicas de paralelização, *Task Parallelism* poderia tornar-se o modelo de paralelismo mais viável para uma gama muito ampla de aplicações, contribuindo para que a programação paralela torne-se mais acessível e vantajosa para muitas equipes de desenvolvimento. É esta razão, em suma, que motiva o presente trabalho a desenvolver um runtime system de *Task Parallelism* de mais alto desempenho com auxílio de *offload* de computação para hardware dedicado.

Objetivos

Sendo assim, o presente trabalho tinha por meta, em suma, implementar um runtime OpenMP 4.0 derivado do Intel OpenMP Runtime (IOMP) que, integrado ao módulo FPGA dedicado à inferência das relações de dependência de dados entre *tasks* a partir de informação a respeito do modo de acesso destas aos seus dados, pudesse oferecer um desempenho superior ao apresentado por runtimes OpenMP 4.0 sem semelhante aceleração.

Desenvolvimento do Trabalho

Visão geral

Para implementar tal runtime, dispúnhamos do acelerador de resolução de dependências em formato bitstream destinado à plataforma Zedboard bem como de descrições de sua interface de registradores e de suas filas de comunicação em memória principal.

A plataforma Zedboard consiste em uma placa de desenvolvimento de sistemas embarcados que conta com um ARM A9 dual-core rodando a 667Mhz, 512MB de memória principal DDR3, interfaces de comunicação Ethernet, USB OTG, USB UART, uma FPGA Xilinx Zynq 7000 além de uma entrada para cartões de memória.

Um aspecto interessante dessa placa de desenvolvimento é o seu suporte a ACP (Accelerator Coherence Protocol), que dá ao acelerador acesso direto aos dados nos caches L1 e L2 do processador, garantindo que se tenha uma latência de acesso aos pacotes trocados através da interface hw/sw muito menor do que a que se poderia verificar com estratégias alternativas baseadas em uso de barramento AXI, message-passing através da memória principal, etc.

Uma vez configurado um servidor ssh em uma instância do Linux Linaro instalada em um cartão de memória para uso na placa, o acesso por ssh à placa tornou o modo padrão de se utilizar esse ambiente de teste.

Com esse ambiente Unix já configurado, a instanciação do bitstream que implementava o acelerador consistia em simplesmente se escrever o seu conteúdo em /dev/xdevcfg através de $ cat acc_bistream.bit > /dev/xdevcfg.

Para permitir o acesso do runtime OpenMP ao acelerador, demandou-se a implementação de dois drivers linux: um, chamado de nexusIO, era destinado ao mapeamento dos registradores de comunicação do acelerador sobre memória de usuário; o segundo, chamado nexusIO_ddr, era destinado ao mapeamento das filas de comunicação com o acelerador sobre o espaço de endereçamento do usuário. Tais filas de comunicação organizavam-se sobre uma região da memória principal que foi reservada exclusivamente para tal fim durante a compilação do *kernel* da distribuição Linux utilizada.

Com esses artefatos já em mãos, o desenvolvimento do restante do *runtime* resumiu-se na construção de duas bibliotecas compartilhadas: uma, chamada **mtsp-bridge**, tinha por objetivo reimplementar as chamadas de função definidas pela API do runtime IOMP, o que lhe permitia conectar-se a aplicações OpenMP 4.0 compiladas para o runtime Intel e receber suas requisições de escalonamento de tarefas; a segunda, chamada **libmtsp**, destinava-se a esconder todos os detalhes do protocolo de comunicação com o acelerador debaixo de uma API de comunicação mais abstrata, facilitando o desenvolvimento de mtsp-bridge.

Interação com aspectos relevantes da arquitetura ARM

A arquitetura ARM define um *weakly-ordered memory consistency model*, o que quer dizer que essa arquitetura dá muito menos garantias de que CPUs trabalhando em um mesmo programa multi-thread terão uma visão consistente do espaço de memória do processo do que arquiteturas como a x86. Isso quer dizer que, caso não sejam empregadas as instruções de ordenamento explícito de memória sempre que necessário, sistemas multi-thread rodando em ARM32 ou ARM64 serão mais propensos a sofrer com problemas de race conditions, leitura de stale data, dead-locks, starvation, do que versões rodando nessas arquiteturas alternativas, que dão garantias muito mais fortes de consistência de memória - ainda que não sejam estritamente *sequentially consistent*.

Sendo assim, mecanismos de passagem de mensagem entre o acelerador e a camada de software que implementamos devem empregar instruções de ordenação explícita de memória ou mesmo de manutenção de cache (cache invalidation, cache cleaning) em diversos pontos.

Um exemplo concreto disso é o mecanismo de submissão de informações a respeito de novas tasks para o escalonador em hardware. Para cada *task* gerada pelas aplicações OpenMP, constrói-se um descritor de metadados contendo informações a respeito dos ponteiros da função e dos parâmetros de tal *task*. Esses descritores são transmitidos entre o runtime e o acelerador através de uma fila circular (*ring-buffer*) mapeada em espaço de usuário ao qual o acelerador tem acesso através do protocolo ACP, que permite que o acelerador leia dados dessa fila diretamente do cache de processador em que ela está armazenada, sem qualquer interação com a memória principal. **A arquitetura do acelerador então** define que este só irá

checar se existem novos elementos na fila de submissão depois que o runtime emitir um sinal de *Doorbell*, que consiste na escrita de um valor qualquer em um certo registrador destinado a tal fim. Por outro lado, a arquitetura ARM não dá nenhuma garantia a respeito da ordem em que o acelerador poderia observar as modificações feitas pelo processador responsável pelas escritas sobre a fila e o registrador de *Doorbell*, de modo que, caso não sejam usadas instruções de ordenamento que garantam a correta ordem de observação desses dois eventos, é altamente provável que durante a execução de um programa real o acelerador observe a escrita sobre o registrador de *Doorbell* antes da atualização da fila de metadados e acabe por consumir *stale data.*

Sendo assim, o runtime implementado emprega instruções de ordenação explícita de memória em diversos pontos em que ela se faz necessária.

Otimizações empregadas

É altamente desejável que todos os núcleos disponíveis para a execução de tarefas geradas por uma aplicação geradora de tasks estejam sempre executando trabalho útil. Sendo assim, supondo um sistema rodando tasks de tempo de execução médio igual a T_et e dispondo de N núcleos, é interessante notar que seria necessário que o runtime OpenMP que distribui tasks para esses núcleos fosse capaz de escalonar uma tarefa a cada T_et / N unidades de tempo caso se quisesse garantir que todos os núcleos estivessem sempre trabalhando. Sendo assim, é fácil notar que cenários em que o número de processadores disponível é alto ou em que a granularidade das tasks é baixa são especialmente onerosos sobre o runtime, que nesses casos deve manter um alto throughput de escalonamento para garantir que todos os processadores sejam bem utilizados. Consequentemente, um runtime de escalonamento de tasks com suporte a offload de resolução de dependências deve ser capaz de manter um alto throughput de comunicação com o acelerador a ele acoplado se quiser lidar bem com esses cenários especialmente desafiadores já que, caso contrário, o overhead de comunicação poderia anular os ganhos devidos à mais rápida resolução de dependências.

Além disso, um runtime com aceleração de resolução de dependências ainda terá de garantir que sua política de escalonamento de trabalho mantenha um bom *load balancing*, o que pode ser um desafio se os processadores disponíveis não estiverem consumindo trabalho de uma única fila de trabalho em comum mas de filas exclusivas. Também é necessário que o mecanismo em software de distribuição de tasks livres para os processadores seja rápido o suficiente para não limitar o throughput de escalonamento.

Sendo assim, descrevem-se agora algumas otimizações implementadas com o objetivo de minimizar os *overheads* de comunicação com o acelerador e favorecer um melhor *load balancing* entre os cores.

Lazy index update das filas com o acelerador

Um observador de uma fila circular é capaz de identificar quantos elementos há em uma fila desse tipo a partir dos seus índices de escrita e leitura. Caso esses dois índices sejam não idênticos, entende-se que há elementos na fila. Também são simples as operações capazes de determinar a posição do mais antigo elemento guardado pela fila, a posição onde deve ser inserido o próximo, qual é o número de itens atualmente armazenados, etc. Supondo então que uma fila de tal tipo seja utilizada para transportar dados entre um par Produtor-Consumidor, entendemos que o consumidor só poderá notar a inserção de novos dados pelo produtor quando puder observar a correspondente atualização do índice de escrita, bem como o produtor só poderá notar o consumo de dados pelo consumidor quando puder observar a devida atualização do índice de leitura. No entanto, para garantir uma garantir um mais alto throughput de comunicação através dessa estrutura de dados, é usualmente interessante que esses índices de escrita e leitura sejam atualizados somente depois que um certo número não unitário de escritas ou leituras, respectivamente, tenha sido feito, posto que a atualização de qualquer desses índices usualmente demandará o emprego de uma instrução de ordenamento explícito de memória e o tráfego de uma linha de cache inteira entre os dois processadores interagindo com a fila, fatos que podem degradar significativamente o desempenho do sistema como um todo. Sendo assim, o runtime OpenMP que no presente momento descrevemos emprega uma política de *lazy-update* dos índices das filas circulares que utiliza para comunicar-se com o acelerador, que se vale dessas considerações para manter um throughput mais alto de comunicação entre as threads ao mesmo tempo que se faz um uso mais consciente do barramento de coerência conectando os processadores.

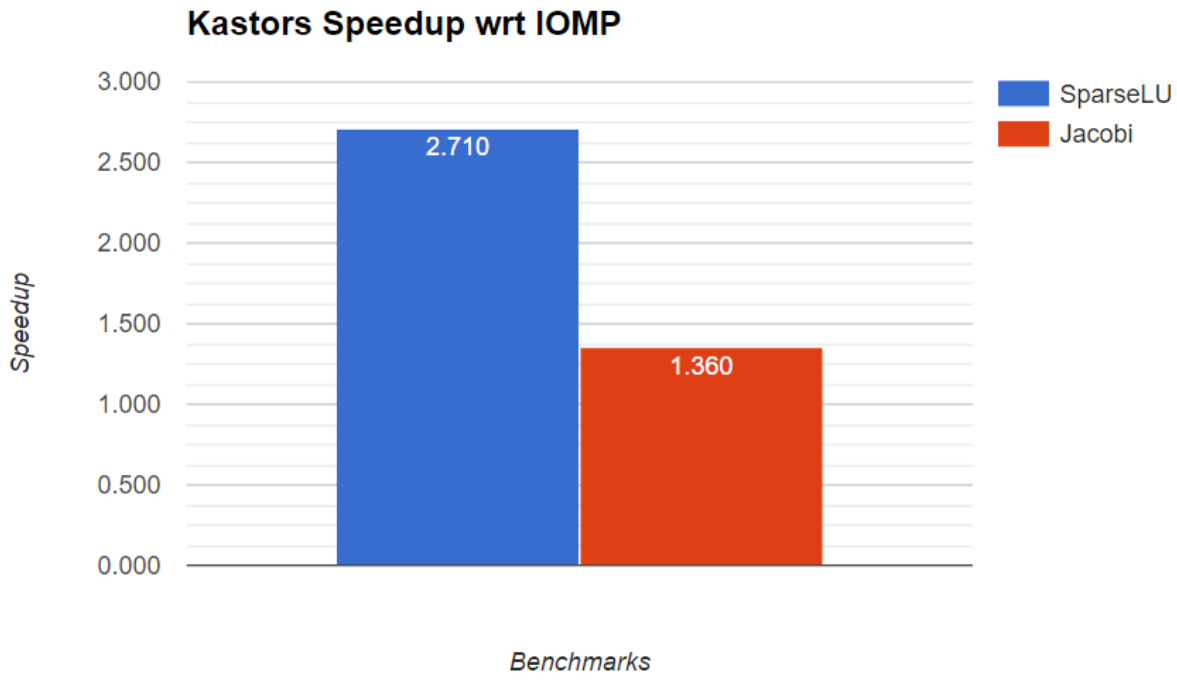
Eliminação de mutexes por uso de Helper Thread

Além disso, procurou-se evitar o uso de mutexes para o recebimento de tarefas livres por parte das threads executoras de trabalho através do emprego das filas circulares de trabalho exclusivas de cada uma dessas threads, que são então alimentadas por uma única thread dotada da capacidade de consumir trabalho da RunQ compartilhada com o acelerador. Isso evita o uso de mutexes pelo fato dessas filas, por natureza, dispensarem o uso dessas primitivas de sincronização.

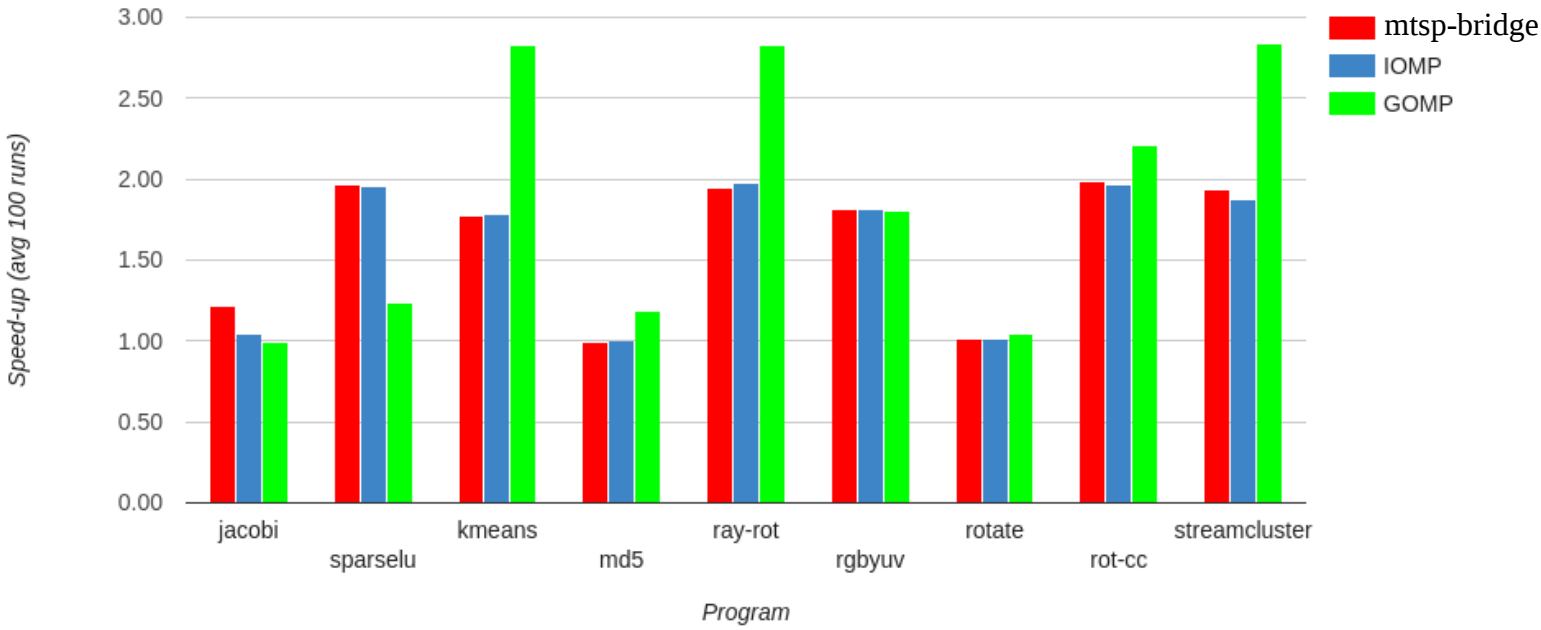
Distribuição de trabalho round-robin

O mecanismo de escalonamento de tasks livres para os núcleos disponíveis emprega uma política de distribuição de tasks à maneira *round-robin* para as filas privadas de trabalho dos cores. Esse algoritmo de escalonamento, embora simples, produz um bom *load balancing* para diversas aplicações, além de implicar em um footprint de tempo computacional muito baixo.

Resultados



mtsp-bridge speedups



Conclusões

A partir dos resultados aferidos, percebemos que, para aplicações geradoras de tasks de baixa granularidade - como é o caso dos benchmarks indicados pela primeira figura, que utilizavam para tal experimento parâmetros de execução que implicavam em baixa granularidade de *tasks*, o runtime com suporte a aceleração de resolução de dependências apresenta um desempenho muito superior ao dos runtimes sem esse recurso. Por outro lado, percebeu-se também – de acordo com a segunda figura – que o runtime com aceleração tende a apresentar desempenho muito similar ao das alternativas em software em cenários de tasks de maior granularidade.

Além disso, é interessante voltar a notar que o runtime OpenMP que desenvolvemos implementa a API de comunicação com aplicações OpenMP com suporte a tasks definida pelo Intel OpenMP Runtime (IOMP). Tal fato traz consigo a grande vantagem de se poder executar aplicações geradoras de tasks tendo por alvo o IOMP com o nosso runtime sem qualquer necessidade de recompilação, o que evita a necessidade de se fazer qualquer modificação sobre a geração de código operada pelo Clang ao interpretar os pragmas OpenMP, o que seria necessário caso quiséssemos definir uma nova API para o runtime.

Por outro lado, isso também significa que alguns aspectos pouco favoráveis a desempenho dessa interface acabaram por ser herdados pelo novo runtime. Por exemplo, de acordo com a interface IOMP, a geração de uma única task envolve a execução de duas chamadas de função, o que é especialmente problemático quando as tasks geradas apresentam baixa granularidade, caso em que o overhead dado por essas chamadas de função poderia limitar consideravelmente o throughput de geração de trabalho.

Sendo assim, com o intuito de se obter uma medida mais próxima do máximo desempenho alcançável com offload de resolução de dependências, seria interessante construir, no futuro, uma interface mais leve para o runtime, que elimine diversos overheads próprios do runtime IOMP. Por simplicidade, essa interface não será baseada na interpretação de pragmas OpenMP, mas em uma API de chamadas de função próprias, à semelhança da interface do Intel TBB. Uma vez avaliados os ganhos de performance dessa alternativa, seria interessante permitir que o Clang pudesse gerar código a partir dos pragmas OpenMP que fossem compatíveis com essa interface.

Referências

[TSC] Y. Etsion, F. Cabarcas, A. Rico et al, *Task Superscalar: An Out-of-Order Task Pipeline* (2010).

[OPM4] OpenMP Architecture Review Board: *OpenMP Application Program Interface 4.0*, (2013)