Proposta de Suporte ao Paralelismo de GPU na SPar

Dinei A. Rockenbach[1], **Dalvan Griebler**[1,2], **Luiz Gustavo Fernandes**[1]

[1]Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Porto Alegre – RS – Brasil

[2]Laboratório de Pesquisas Avançadas para Computação em Nuvem (LARCC), Faculdade Três de Maio (SETREM), Três de Maio – RS – Brasil

dinei.rockenbach@edu.pucrs.br,dalvan.griebler@acad.pucrs.br

Resumo. *As GPUs (*Graphics Processing Units*) têm se destacado devido a seu alto poder de processamento paralelo e sua presença crescente nos dispositivos computacionais. Porém, a sua exploração ainda requer conhecimento e esforço consideráveis do desenvolvedor. O presente trabalho propõe o suporte ao paralelismo de GPU na SPar, que fornece um alto nível de abstração através de uma linguagem baseada em anotações do C++.*

1. Introdução e Caracterização

A velocidade na frequência de operação dos núcleos dos processadores modernos está praticamente estagnada desde o início do século, principalmente devido à chamada *power wall*, que é a incapacidade de dissipar o calor resultante por aumentos no consumo energético induzidos pelo aumento na frequência do processador [McCool et al. 2012]. Essa estagnação marcou o início de uma era de ubiquidade do hardware paralelo, que foi a resposta dos fabricantes ao desejo dos desenvolvedores e usuários por máquinas mais rápidas. Porém, essa transição para processadores *multi-core* exigiu uma correção de rumo no desenvolvimento de software, até então acostumado a receber os benefícios de máquinas exponencialmente mais rápidas sem alterar o código-fonte.

Para auxiliar na transição para o desenvolvimento de software paralelo, tanto a indústria [Reinders 2007] quanto a academia [Thies et al. 2002, Aldinucci et al. 2009] desenvolveram várias ferramentas. Nos últimos anos, as pesquisas em *algorithm skeletons* e *design patterns* têm convergido no sentido de oferecer padrões de programação paralela e possibilitar o desenvolvimento de ferramentas focadas na aplicação de soluções já definidas para estes padrões [McCool et al. 2012, Mattson et al. 2004]. Dentre essas ferramentas está a SPar [Griebler et al. 2017], uma linguagem de domínio específico (*Domain-Specific Language* ou DSL) focada no paralelismo de *stream*. A SPar oferece atributos C++11 que podem ser inseridos em regiões paralelizáveis do código através de anotações. Além disso, existe um compilador que efetua transformações *source-to-source*, inserindo automaticamente chamadas ao framework FastFlow [Aldinucci et al. 2009] para explorar o paralelismo de *stream* através da geração eficiente dos padrões *pipeline* e *farm*, ou a composição de *pipeline* com *farm*(s).

Na história recente, o desempenho máximo em cálculo de ponto flutuante da arquitetura *many-thread*, cuja principal representante é a GPU (*Graphics Processing Unit*), têm se mantido em torno de 10 vezes o desempenho das CPUs (*multi-core*) [Kirk and Hwu 2016]. Porém, atualmente não é possível explorar o paralelismo dessas arquiteturas heterogêneas

na SPar de forma automática, pois o seu compilador suporta apenas a geração de código para sistemas *multi-core*. No entanto, a exploração do paralelismo de *stream* em sistemas *multi-core* pela SPar combinado com o paralelismo de dados em GPU utilizando as linguagens CUDA e OpenCL já foi testado em [Stein et al. 2019], obtendo resultados encorajadores. Somando isso à capacidade de oferecer alta produtividade mantendo um desempenho muito similar a outros *frameworks* [Griebler et al. 2018], propõe-se a inclusão de suporte ao paralelismo em GPU a fim de impulsionar a classe de aplicações de processamento de *stream* com características altamente paralelas, tais como modelos de redes neurais profundas (*Deep Learning*). Desta forma, o objetivo é permitir que o programador expresse o paralelismo de dados nativamente por meio das anotações da SPar, bem como permitir a geração de código-fonte paralelo de forma automatizada e abstrata.

Vários trabalhos visam à exploração do paralelismo em GPU através de uma linguagem com um nível de abstração maior do que ferramentas como CUDA e OpenCL. A biblioteca SkelCL [Steuwer et al. 2011] e os *frameworks* SkePU 2 [Ernstsson et al. 2018] e FastFlow [Aldinucci et al. 2016] são exemplos desses esforços. Baseados em *algorithmic skeletons*, eles facilitam a programação para GPUs mas não isentam o desenvolvedor do trabalho de definir os *kernels* que serão executados na GPU. O StreamIt [Thies et al. 2002] é uma linguagem específica de domínio para aplicações de *stream*, porém os níveis de abstração e invasão de código são radicalmente diferentes daqueles oferecidos pela SPar. Compiladores para transformar o código StreamIt em código CUDA foram propostos por [Udupa et al. 2009] e [Hormati et al. 2011], porém nenhuma das soluções apresenta o nível de abstração provido pela SPar ao programador de aplicações. De fato, *frameworks* como o FastFlow e o SkePU 2 e bibliotecas como o SkelCL são encarados como potenciais *runtimes* para a SPar.

2. Suporte ao Paralelismo de GPU na SPar: Pesquisa em Andamento

A linguagem SPar é composta pelos atributos `ToStream` e `Stage` que identificam regiões de *streaming* no código e etapas de computação (que podem ser estágios ou etapas que se assemelham a uma linha de montagem). Além disso, existem os atributos auxiliares `Input` e `Output` que, por sua vez, identificam variáveis de entrada e saída (elementos do *stream*), enquanto que o atributo auxiliar `Replicate` demarca um `Stage` que pode ser replicado em paralelo. A SPar atualmente gera código paralelo baseado nos padrões *pipeline* e *farm* para explorar o paralelismo de *stream*. Conforme demonstrado em [Stein et al. 2019], o paralelismo de *stream* pode ser explorado em conjunto com o paralelismo de dados, utilizando uma composição de padrões paralelos adequados a cada um dos tipos de processamento. As GPUs são mais indicadas ao paralelismo de dados, portanto é preciso que a geração de código no compilador da SPar suporte também padrões adequados para o paralelismo de dados, tais como *map* e *reduce*.

A proposta do trabalho envolve alterar a sintaxe da SPar adicionando dois atributos, `Pure` e `Batch`, ao conjunto já existente. O atributo `Pure` poderá ser utilizado como identificador (ID) ou auxiliar (AUX) em conjunto com o atributo `Stage`. Ele identificará que um bloco de código é uma função pura, ou seja, cuja saída depende apenas na entrada, sem dependências ou efeitos colaterais externos [McCool et al. 2012]. A presença desse atributo indicará ao compilador que o padrão map pode ser gerado utilizando o bloco de código. O atributo `Batch`, por sua vez, é utilizado apenas em conjunto com o atributo `Pure` e indica que os itens do map devem ser agrupados para realizar o processamento em

lotes.

Regras de transformação serão criadas e implementadas no compilador para que seja possível gerar os padrões paralelos adequados ao paralelismo de dados a partir da presença do atributo `Pure`, que indica a possibilidade de fazer o *offload* do trecho de código para aceleradores como GPUs. Nesse sentido, o compilador analisará o código-fonte dentro de um `Stage` com `Pure` para aplicar padrões paralelos como *map* e *reduce*, além de identificar o código que possa ser enviado como *kernel* à GPU. Um exemplo da proposta de código sequencial anotado pode ser visto no Código 1.

```
int prime_number ( int n ) {
  int total = 0;
  [[spar::ToStream,spar::Input(total,n),
    spar::Output(total)]] {
    [[spar::Stage,spar::Pure,spar::Batch(size),
    spar::Input(total,n),spar::Output(total),
    spar::Replicate()]]
    for (int i = 2; i <= n; i++ ) {
      int prime = 1;
        for (int j = 2; j < i; j++){
          if ( i % j == 0 ) {
            prime = 0;
            break;
          }
        }
        total = total + prime;
      }
    }
  }
  return total;
}
```

Código 1. Cálculo de números primos com SPar.

A escolha do nome dos atributos leva em consideração a premissa da SPar de abstrair a arquitetura do desenvolvedor, bem como trabalhos futuros na linguagem. A palavra *pure* já é utilizada em linguagens como Fortran para identificar funções sem efeitos colaterais e também é uma palavra com significado para programadores inexperientes. Por sua vez, *batch* também é uma palavra comum tanto no ambiente de programação quando fora dele. Está sob avaliação a possibilidade de permitir ao desenvolvedor especificar, através de um parâmetros, a possibilidade de explorar o paralelismo em CPU com base nos mesmos atributos. A proposta atual, contudo, engloba oferecer suporte apenas à GPU.

A criação e implementação das transformações *source-to-source* envolverá uma série de etapas, que podem ser resumidas em: (1) etapa de *parse* e análise do código-fonte, em que a entrada é o código-fonte e a saída é uma *Abstract Syntax Tree* (AST); (2) etapa de análise semântica da AST, em que são validadas as regras de semântica do código e dos atributos da SPar; (3) etapa de transformação da AST, em que o código é modificado e ocorre a inclusão das estruturas necessárias para exploração do paralelismo; (4) geração do código a partir da AST transformada, novamente com vistas ao padrão ISO C++; (5) compilação do código final e geração do arquivo binário compilado.

A maioria das implementações da presente proposta estão contidas nas etapas (2) e (3), onde são feitas a validação e alterações referentes aos atributos presentes no código-fonte, porém não estão descartadas alterações nas outras etapas do compilador da SPar. A definição das regras de transformação do código-fonte anotado nos padrões paralelos propostos está em andamento e os resultados parciais têm sido promissores, apresentando speedups de $39\times$ com o Código 1 e $78\times$ com um algoritmo simples de multiplicação de matrizes. A validação foi feita aplicando-se manualmente as regras propostas.

Após a definição das regras de transformação, o passo seguinte é incluir as regras no compilador para que este gere chamadas às rotinas das bibliotecas CUDA e OpenCL

no código anotado. A geração de código CUDA e OpenCL poderá ser controlada através de *flags* do compilador.

A principal contribuição deste trabalho foi estender e propor dois novos atributos para a linguagem SPar (`Pure` e `Batch`), que indicam a presença de código sem restrições para a exploração do paralelismo massivo em coprocessadores. O compilador da linguagem será alterado para gerar código paralelo automático com base nesse e nos outros atributos de alto nível da linguagem, inferindo informações como tipo e tamanho dos dados com base na análise da AST.

Referências

Aldinucci, M., Danelutto, M., Drocco, M., Kilpatrick, P., Misale, C., Peretti Pezzi, G., and Torquati, M. (2016). A parallel pattern for iterative stencil + reduce. *Journal of Supercomputing*, pages 1–16.

Aldinucci, M., Danelutto, M., Meneghin, M., Torquati, M., and Kilpatrick, P. (2009). Efficient streaming applications on multi-core with FastFlow: The biosequence alignment test-bed. In *International Conference on Parallel Computing (ParCo)*, pages 273–280, Lyon, France.

Ernstsson, A., Li, L., and Kessler, C. (2018). SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming*, 46(1):62–80.

Griebler, D., Danelutto, M., Torquati, M., and Fernandes, L. G. (2017). SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):1740005.

Griebler, D., Hoffmann, R. B., Danelutto, M., and Fernandes, L. G. (2018). High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. *International Journal of Parallel Programming*, pages 1–19.

Hormati, A. H., Samadi, M., Woh, M., Mudge, T., and Mahlke, S. (2011). Sponge: Portable Stream Programming on Graphics Engines. *SIGPLAN Not.*, 47(4):381–392.

Kirk, D. B. and Hwu, W.-m. W. (2016). *Programming Massively Parallel Processors*. Morgan Kaufmann.

Mattson, T. G., Sanders, B. A., and Massingill, B. L. (2004). *Patterns for Parallel Programming*. Software Patterns Series. Pearson Education.

McCool, M., Robison, A. D., and Reinders, J. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann.

Reinders, J. (2007). *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Series. O'Reilly Media.

Stein, C. M., Griebler, D., Danelutto, M., and Fernandes, L. G. (2019). Stream Parallelism on the LZSS Data Compression Application for Multi-Cores with GPUs. In *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Pavia, Italy. IEEE.

Steuwer, M., Kegel, P., and Gorlatch, S. (2011). SkelCL - A portable skeleton library for high-level GPU programming. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1176–1182. IEEE.

Thies, W., Karczmarek, M., and Amarasinghe, S. (2002). StreamIt: A Language for Streaming Applications. In Horspool, R. N., editor, *Compiler Construction*, pages 179–196, Berlin, Heidelberg. Springer.

Udupa, A., Govindarajan, R., and Thazhuthaveetil, M. J. (2009). Software pipelined execution of stream programs on GPUs. In *Proceedings of the 7th International Symposium on Code Generation and Optimization (CGO)*, CGO '09, pages 200–209, Seattle, WA, USA. IEEE.