UERJ – UNIVERSIDADE DO ESTADO DO RIO DE JANEIRO
IME – INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
Ccomp – Programa de Pós-Graduação em Ciências Computacionais
Tópicos Especiais em Inteligencia Computacional e Meta-heurísticas
Prof. Igor Machado
Aluno Paulo José de Alcantara Gimenez

2o Trabalho

Uma abordagem Dataflow com CPU-GPU para Exploração de Vizinhança Múltipla em Otimização Combinatorial

1. Introdução

Problemas difíceis (*NP-Hard*) podem em abordagens computacionais convencionais levar dias, anos, décadas ou séculos para encontrar uma solução ótima. Assim muitos estudos tem explorado uso de algoritmos que tentem obter solução aproximadamente ótima, notadamente as metaheuristicas. Em concomitância, com o advento das tecnologias dos múltiplos núcleos de processamento, as pesquisas com a aplicação de computação paralela, que leva a readequação dos algoritmos sequenciais tradicionais, também tem sido explorados. O uso de múltiplas CPUs tem sido explorado mesmo nas áreas de otimização, como arquitetura performática baseado nos coprocessadores para redução do tempo de solução por explorar adequadamente as rotinas *CPU-bound*.

Muitos estudos e evoluções na computação paralela tem ocorrido ao longo dos anos, para tornar mais transparente a arquitetura e sobretudo a programação, adaptando os algoritmos sequências existentes. Mas, apesar desses avanços, a programação *multi-threaded* não é trivial e é propensa a erros, devido principalmente a questão de *race-condition* e cenários complexos para análise e verificação, conforme visto em Outerhout [Outerhout, 1996]. Nesse cenário, a tecnologia GPU (*Graphic Processing Unit*) que é baseada na arquitetura paralela massiva, foi primeiramente usado para suporte as funções gráficas de aplicativos e posteriormente utilizados para melhorar o desempenho de soluções aplicadas à ciência da computação – sendo rebatizada como GPGPU (*General Purpose* GPU), em face de seu melhor custo-benefício em relação as CPUs. Contudo, as plataformas e ferramentas para GPU ainda não estão consolidadas, oferecendo comportamentos e resultados específicos devido dependências específicas da versão do dispositivo utilizado, o que pode tornar inviável a execução em ambientes paralelos heterogêneos por software ou hardware. Assim, frequentemente algoritmos propostos para problemas específicos tendem a ser também específicos ao ambiente implementado, seja para um determinado dispositivo GPGPU, versão de ferramentas (CUDA X., OpenCL a.b etc) ou instalação do SO, bem como a combinação da arquitetura multi CPU-GPU.

A programação no modelo Dataflow, onde há independência do ciclo ou conjunto de instruções em execução, considerando sempre o fluxo dos dados e sua interdependência, é capaz de fornecer paralelismo de processamento genérico sem a maior complexidade envolvida no gerenciamento de *threads*, dependência funcional da arquitetura do hardware CPU-GPU, considerando que cada instância do ambiente dado seus recursos computacionais seria elegível para cada parte do processamento requisitado sobre os dados em determinada transição. A programação assim, pode usar a arquitetura de *Von Newman* ou mesmo a arquitetura *DataFlow*, sendo necessário na primeira mapear as funções específicas por recurso, e na última a existência da representação do recurso na implementação do modelo utilizada. E através do uso de uma biblioteca e plataforma minimalista do modelo Dataflow como é o caso da Sucuri feita em linguagem Python pretende-se simplificar a adoção desse paradigma em alto nível de abstração.

Neste presente trabalho, utilizamos a biblioteca Sucuri para multicomputadores segundo arquitetura de *Von Newman* e utilizando MPI (*Message Passing Interface*), e exploramos as possibilidades de adequação e ajuste mínimo em uma implementação de metaheurística para execução em GPU através de programação em CUDA (*Compute Unified Device Architecture*, é uma extensão para a linguagem de programação C, a qual possibilita o uso de computação paralela), considerando as variações e possíveis expansões da plataforma Sucuri para ser mais aderente aos recursos da GPU através de CUDA pela sua implementação em *Python* denominada *PyCuda*. Problemas de compartilhamento de dados em memória entre os múltiplos processos em dispositivos GPU em máquinas distintas, combinação dos processos CPU+GPU e quais etapas do algoritmo são elegíveis para CPU e/ou GPU.

2. A metaheurística

A heurística Busca Local e uma das mais antigas técnicas heurísticas, aplicada para encontrar e melhorar a qualidade de uma solução candidata inicial (s0), através de mudanças sistemáticas dessa solução pela descoberta de uma direção descendente com um vizinho N(s0) selecionado, procedendo o valor minemo (ou máximo) de uma função f(s0) com o vinho N(s0) na mesma direção, até que atinja uma solução ótima local [Coelho, 2016][Coelho, 2017]. Normalmente faz parte de muitas metaheuristicas. Por explorar variações controladas e programadas de soluções candidatas em cada transição, é um algoritmo que pode ser beneficiar da computação paralela para acelerar a localização da solução ótima local. Também com o paralelismo pode-se explorar diversas linhas de buscas locais.

A exploração de vizinhanças para determinar a solução ótima global, ou a mais próxima disso, para um problema *NP-Hard* de otimização mínima (ou máxima) são normalmente tratados pela classe de metaheuristicas de trajetória, e, nessa classe o método VNS (*Variable Neighborhood Search*) tem especial papel. A VNS foi proposta por *Mladnovic* e *Hansen* em 1997 para resolver esses problemas explorando as distâncias de vizinhança da solução candidata corrente, movendo dessa para uma nova se somente se houver melhora na solução, de forma sistemática modifica a vizinhança primeiro pela descida para encontrar um ótimo local e depois efetua uma pertubação para sair do vale (valor mínimo) correspondente. A busca local é aplicada repetidamente para encontrar a solução ótima local em uma vizinhança.

Muitas vari acoes e extensões à VNS tem sido propostos para campos aplicados de problemas específicos, bem como tem sido explorado o uso de paralelismo para esse fim. Vale destacar que as estratégias clássicas de busca locais usadas para esse fim são: (a) *best improvement*, (b) *First Improvement.* A metaheurística utilizada no presente trabalho é uma implementação inspirada na VNS proposta por [Coelho, 2016], que em vez de usar as abordagens clássicas de seleção aleatória ou as citadas anteriormente, apresenta uma nova denominada *multi improvement.* Os algoritmos básicos dessas abordagens é visto na Tabela 1 a seguir.

<table>
<tr><th colspan="3">Algorítmos (foco no valor mínimo)</th></tr>
<tr><th>Best Improvement</th><th>First Improvement</th><th>“classical” Neighborhood Change</th></tr>
<tr>
<td>Function BestImprovement(x)
1: repeat
2: x' ← x
3: x←argmin_{f (y)}, y∈N(x)
4: until (f (x) ≥ f (x'))
5: return x</td>
<td>Function FirstImprovement(x)
1: repeat
2: x' ← x; i←0
3: repeat
4: i←i+1
5: x←argmin{ f (x), f (x^i)}, x^i ∈ N(x)
6: until (f (x) < f (x^i) or i = |N(x)|)
7: until (f (x) ≥ f (x'))
8: return x</td>
<td>Function NeighborhoodChange (x, x', k)
1: if f (x') < f(x) then
2: x ← x' // Make a move
3: k ← 1 // Initial neighborhood
4: else
5: k ← k+1 // Next neighborhood</td>
</tr>
<tr><td colspan="3">Basic VNS</td></tr>
<tr><td colspan="3">Function VNS (x, kmax, tmax);
1: repeat
2: k ← 1;
3: repeat
4: x' ←Shake(x, k) /* Shaking */;
5: x'' ← FirstImprovement(x') /* Local search */;
6: x ←NeighbourhoodChange(x, x'', k) /* Change neighbourhood */;
7: until k = k_max ;
8: t ←CpuTime()
9: until t > t_max ;</td></tr>
</table>

Tabela 1 – Algoritmos clássicos de VNS e Busca Local

3. Dataflow

O modelo de programação *Dataflow* foi proposto na tese de doutorado de *Bert Southerland* (Sutherland) a mais de 50 anos, mas a arquitetura computacional adotada na indústria e academia ate recentemente não provia o suporte de eficiência necessária, até o surgimento das CPU MultiCores, GPUs, FPGA etc, onde o modelo de programação orientado a instruções padece com controles de múltiplas linhas de processamento paralelo, difícil de rastrear e tratar falhas.

No modelo de programação Dataflow internamente as aplicações são representadas em termos de gráficos direcionados (início → fim) como um diagrama visual de fluxo de dados (Boldt, 2015). Os componentes básicos para construção da aplicação são conjuntos de nós, também conhecidos como blocos, o grafo que representa o fluxo lógico dos dados, e a operação que representa a função especializado do nó. Tais nós possuem portas de entradas e/ou saídas para fluxo dos dados como argumento. Os nós podem ser de entrada, processamento ou sumidouros nesse fluxo de informação na aplicação. Os nós são conectados por arestas direcionais dentro do grafo que define o fluxo da informação entre esses mesmos nos.

Conforme (Boldt, 2015), a programação em Dataflow fornece vantagens naturais sob o ponto de vista da disponibilidade de linguagens de programação visuais, o que facilita a compreensão e adoção, além do suporte implícito à concorrência, uma vez que cada no é um bloco de processamento independente, não existindo efeitos colaterais e um modelo de execução onde o node é executado assim que todos os dados necessários estejam disponíveis. Isto é a essência do modelo de execução Dataflow, eliminando mecanismos artificiais ou programados para tratar concorrência, permitindo atualmente explorar melhor a CPU multicores e paralelismo, enquanto reduz a complexidade de desenvolvimento associado.

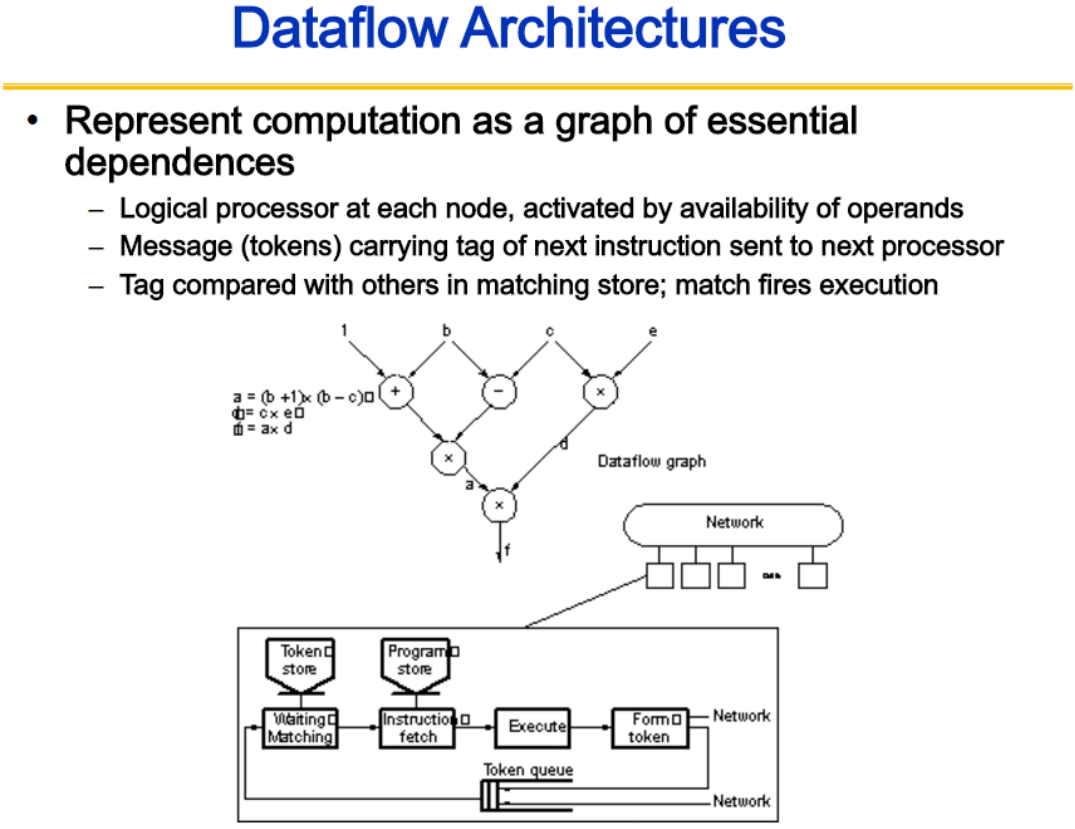

Figura 1 – Representação e conceituação computacional da Arquitetura Dataflow [Patterson, 2015]

Na arquitetura segundo o modelo Dataflow, todo o fluxo de processamento é executado seguindo o grafo proposto para a aplicação desejada, podendo ter um ou mais nós de entrada, um ou mais nós de término e nós de processamento especializados entre eles ligados direcionalmente entre si como arestas do grafo [Boldt, 2015]. Essas arestas introduzem a dependência entres os nós materializadas através das conexões das portas de saídas de um ou mais nos nas de entrada de outros nos sucessivamente, podendo haver sobreposição ou repetição. Tal comportamento, estrutura e representação pode ser visto na Figura 1 acima [Patterson, 2015].

No campo da computação científica, onde exitem diversos algoritmos e métodos de calculo e funções conhecidos, e as abordagens de aplicação são orientadas a assuntos e baseados no processamento e fluxo orientado de informações específicas, a programação em Dataflow se beneficia por poder encapsular em nós especifico para os algoritmos e métodos, o grafo para desenhar o fluxo da informação orientando do problema a solução. Nesse campo de atuação, a linguagem Python tem ampla disseminação e uso, não só pelos especialistas em computação mas também pelos especialistas de diversas áreas da ciência. Nesse contexto introduzimos o uso da Sucuri [Alves, 2014][Sena, 2015], uma plataforma minimalista de programação textual em Dataflow na linguagem Python.

3.1. Sucuri

A Sucuri [Alves, 2014] é uma biblioteca totalmente implementada em Python que permite programação Dataflow em alto nível com suporte implícito a programação paralela através do MPI. Possui quatro componentes básicos: *DFGraph*, *Scheduler*, *Node* e *Worker* para representar e executar o comportamento de recursos computacionais em uma arquitetura *Von Newman*.

O componente *DFGraph* mantém a relação de conjuntos de *Nodes*, cada um contendo (a) lista de operandos recebidos prontos e esperando para correspondência, (b) a função a ser executada quando o nó receberem todos os operandos necessários, (c) lista de destinação que serão usados para enviar a saída de um no para outros nos que dependerem deles, (d) atributos específicos. O modelo de grafo é usado para facilitar a programação paralela.

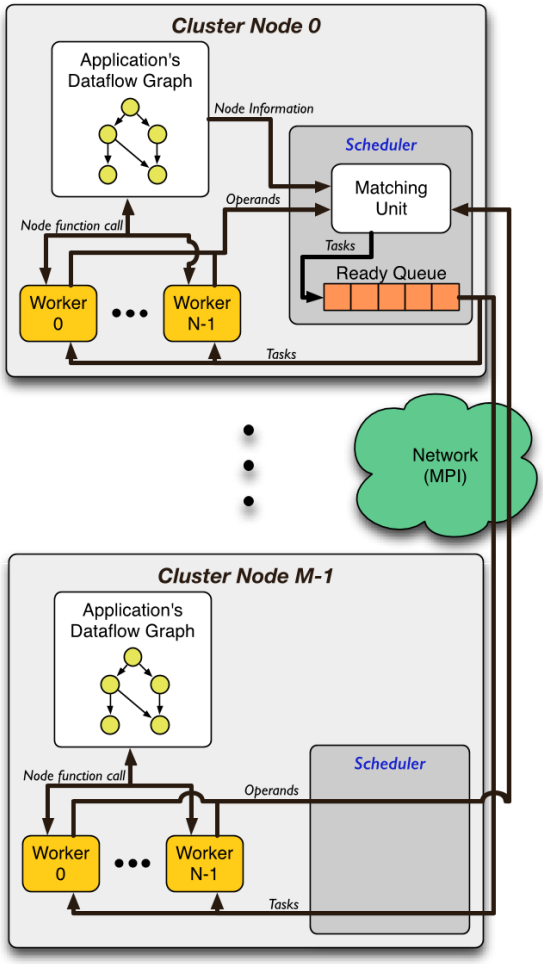

Figura 2 – Representação do modelo de execução Dataflow da Sucuri [Alves, 2014][Sena, 2015]

Cada nó representando pelo componente *Node* expressa um objeto de tarefa de execução concorrente, onde a operação em si é representada pelo componente *Oper* e a tarefa através do componente *Task*. O *Node* contém um método denominado *pin* que é usado para atribuir um *Node* a um determinado *Worker*, fazendo com que somente aquele *Worker* executará aquele *Node*. Para execuções recursivas ou em laços de repetição, há a tipificação especial *TaggedValue* para rotular cada dado em cada iteração ou nível. A representação da operação é dada pela combinação do *Worker* com *Node* e *Task* junto aos argumentos passados. A biblioteca prove ainda uma camada de abstração que permite o mesmo código executar em um multi-CORE ou em um cluster de multi-CORE. O grafo é replicado em todos os nós de um Cluster mas apena o grafo no nó 0 (principal) pode receber operandos do Scheduler principal.

Para simplificação geral de implementação de aplicações na plataforma Sucuri, alguns nós especializados de propósito geral estão disponíveis: *Feeder* que é um no de entrada para representar os argumentos constantes ou dados iniciais; *Source* que representa a entrada de dados externos, normalmente a massa de dados para processar. Como apresentando em [Sena, 2015] é possível construir templates e extensões para propósitos específicos funcionais ou de recursos de computação. Essas especializações de

implementação para um caso particular de aplicação, normalmente se dá pela extensão do componente DFGraph para representar e facilitar um determinado modelo de fluxo dos dados, ou pela particularização funcional de Node para encapsular um comportamento de algoritmos ou métodos de calculo, podendo gerar um repositório de Node especializados para uso em demais aplicações. Para utilização em arquiteturas computacionais distintas, tais como as que possuam recursos de hardware especiais tais GPU, FPGA ou Xeon-Phi, pode-se especializar o Scheduler, sobretudo para processamento distribuído hierárquico ou vertical, ou o Worker para representar uma instância ou Thread de recursos computacional.

4. Experimento

4.1. O cenário problema

O problema do caixeiro viajante (TSP) pode ser expresso por dado um conjunto finito de cidades expressas em termos de custo de distâncias entre cada uma, deseja-se o melhor caminho (valor mínimo) entre duas cidades qualquer, isto é um caso clássico de problema "*short path*". Como o conjunto de cidades pode ser muito grande, o objetivo e encontrar em um espaço de tempo computacionalmente viável uma boa opção de trajeto. Normalmente considera-se o trajeto circular, isto e, computar a distância da cidade de destino de volta a cidade de partida diretamente. Todas as cidades têm trajeto direto as demais cidades, isto e, o grafo e completo, por simplificação.

O problema de latência mínima é uma variação do TSP e pode ser formalmente enunciado como dado um conjunto de n pontos, uma matriz simétrica de distância (dij) e uma viagem que visita os pontos em alguma ordem, deixe a latência de um ponto p ser a duração do passeio desde o ponto inicial até ponto p; a latência total do passeio é a soma das latências de todos os pontos, e desejamos encontrar o passeio que minimiza a latência total, sendo o interesse particular o caso em que a matriz de distância satisfaça a desigualdade do triângulo.

As trocas de vizinhança, ou caminho, podem ser executadas por alguns algoritmos conhecidos, como swap, 2-opt ou oropt-2, dentre outros, onde o objetivo é causar uma pertubação controlada na solução de visita candidata corrente.

4.2. Metaheurística aplicada

Foi utilizado o método da estratégia Multi Improvement [Coelho, 2017] inspirado na meta-heurística VNS para buscar a melhor trajetória para o problema, sob o ponto de vista da minimização. Com o uso de tal método é possível explorar espaço de busca em paralelo por diversos vizinhos com uso de diversos algoritmos de troca e vizinhança (*NeighborhoodChange*). Permite localizar uma aproximação boa para a solução ótima global. A proposta de referência [Coelho, 2017] representa o mapa de trajetos como uma estrutura especial denominada *MLProblem* e o conjunto de soluções em uma estrutura especial *MLSolution* e faz uso de estruturas, blocos e instruções especiais do *Kernel* GPU, através da organização das GPU Threads em blocos, onde cada uma processa um movimento simples e apenas a melhor solução é evoluída no bloco.

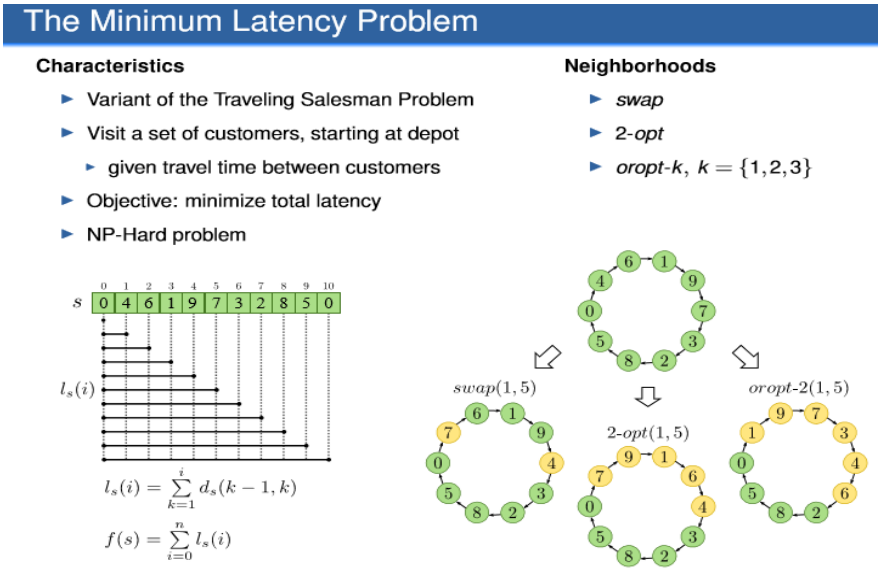

Figura 3 – Qualificação do problema e uso da metaheurística de [Coelho, 2017]

4.3. Experimento

Em modelo de execução Dataflow, geralmente há muitos PE (Process Executor) e cada um executa várias instruções de um programa de Dataflow inteiro, os dados são trocados via rede quando o ambiente for cluster, e as instruções contém os destinos que são acionados quando o PE executa e envia resultados. [Shen et al ,2017]. Na arquitetura implementada pela Sucuri, o roteamento é feito pelo escalonador Scheduler e a eficiência do roteamento e da associação com o recurso GPU é essencial para o desempenho. Conforme [Shen et al, 2017] temos três características principais na transferência de dados em Dataflow: (a) cada instrução tem múltiplos consumidores (destinos); (b) quando houver múltiplos núcleos a transferência somente ocorre quando houver falta de cache; (c) os atrasos nas transferências de dados tem um impacto significativo no desempenho. Cada instrução produz um resultado e este é transferido via rede, exceto quando estiverem no mesmo PE, então, atrasos na transferência dos dados entre os PEs, e assim entre os Workers dos PEs, comprometem o desempenho geral da solução nesse modelo e portanto na implementação feita com a Sucuri.

Uma outra questão interessante é que podem haver limitações de instruções que o um PE suporte simultaneamente, ou mesmo que algumas instruções somente possam ser executadas em determinados Pes, como é o caso das instruções em CUDA para GPU. Nesse caso, precisamos associar a função implementada em CUDA/GPU a ser executada a determinado PE e por conseguinte restrita a Workers específicos. Como para executar instruções no contexto de um dispositivo GPU é necessário um Host Thread CPU associado ao processo na GPU, faz-se necessário adaptar essas características na Sucuri.

Como na implementação da metaheurística de VNS com *Multi Improvement* é necessário distribuir os dados entre os nós que processarão código em CUDA/GPU, faz-se necessário especializar o Node para suportar comportamento adequado à GPU. Também é necessário que seja controlado a avaliação se pode ser efetuada a troca de vizinhança ou não nas funções para esse fim (swap, 2-opt, 3-opt, oropt-2).

Assim na proposta de trabalho presente a Sucuri foi adaptada para suporte a execução de código CUDA/GPU afinizado aos PEs que possuam esse recurso segundo a ordem 0..N do máximo de GPUs disponíveis por instância no cluster. Isto é possível através dos seguintes ajustes:

- GPUScheduler estendendo Sucuri:Scheduler com um método de InitializeWorkers que instância o Worker especial para GPU denominado GPUWorker.
- DFGraphGPU estendendo Sucuri:DFGraph para suporte a quantidade máxima de dispositivos GPU (atributo gpuTargetMax) no ambiente cluster por PE. São nominados de 0 a N, sendo N <= a quantidade máxima definida.
- GPUWorker estendendo Sucuri:Worker para tratar em caso de presença de GPU as tarefas e funções em PyCuda/CUDA. Capaz de detectar em DFGRaphGPU o atributo “gpuMaxTarget” e assim instanciar a host thread CPU associada a execução de comandos no contexto do dispositivo GPU presente no PE.
- GPUHostThread descendendo de Thread, responsável por inicializar o dispositivo e contexto GPU associado a essa thread host.
- FlipFlop descendendo de Sucuri:Node, responsável por avaliar entre os argumentos da solução corrente com a última solução avaliada como melhor e selecionar a próxima solução melhor. Preserva o estado da última solução melhor através de sua própria realimentação no fluxo de dados.
- GPUNode estendendo Sucuri:Node para armazenar o contexto/id do dispositivo GPU associado.

Cabe ressaltar que no caso das extensões à Sucuri, todas funcionam no modo de compatibilidade, isto é, caso não tenha sido fornecida informação sobre o uso de GPU (atributo usingGPU em GPUNode) o processamento fluirá normalmente como CPU.

O impasse restante é a movimentação dos dados entre os PE e o tratamento adequado no GPUWorker para não impactar a performance. Nesse caso, usa-se uma solução de “Fork-Join” [Sena, 2015] para distribuir os dados de solução corrente e matriz de problema. A solução em grafo proposta pode ser vista em Figura 4.

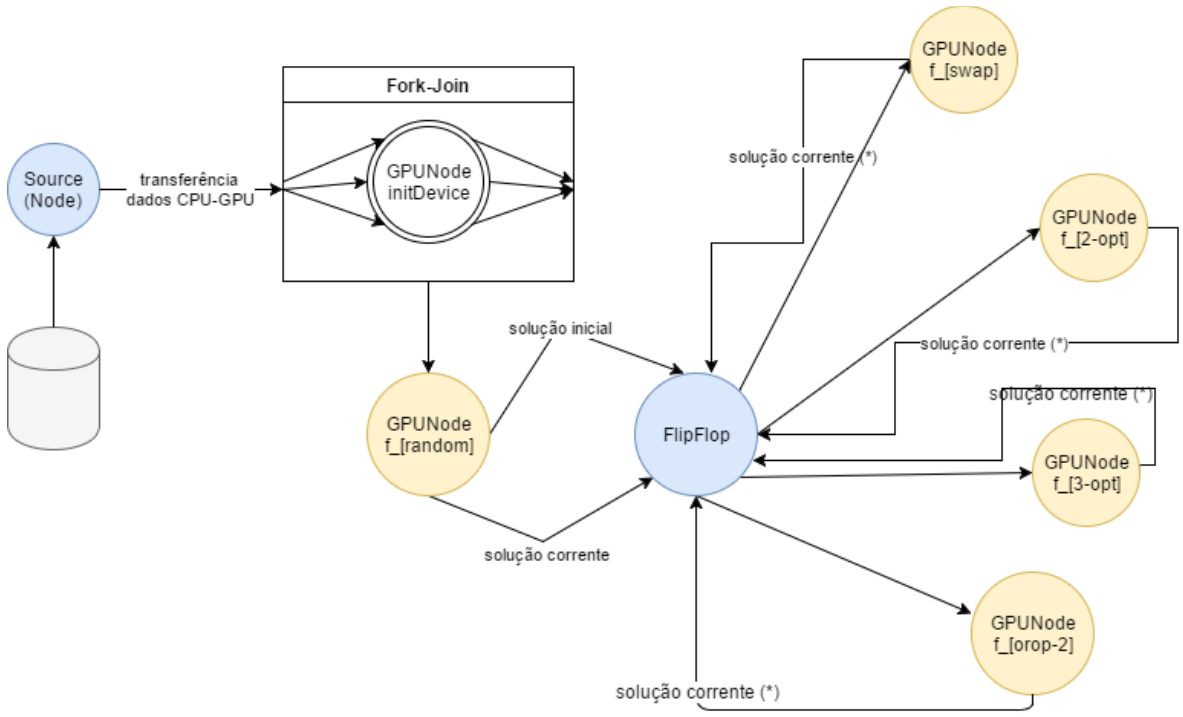

Figura – Grafo de solução para a Multi Improvement balanceando processamentos específicos CPU e GPU e transferência dos dados.

5. Conclusão e Trabalhos futuros

Como desdobramento de trabalho futuro, estabelecer mecanismos estatísticos para avaliar o desempenho das diversas linhas para implementação da arquitetura híbrida em relação a implementação Dataflow proposta na seção 4 anterior, e explorar tais implementações. Considerar também a possibilidade de subgrafos, conforme [Sena, 2015] para tratar o escalonamento das tarefas conforme afinidade com os recursos CUDA/GPU e CPU. Como arquitetura híbrida implementar soluções onde parte do fluxo de processamento é por instrução interagindo com o fluxo de dados da Sucuri, e também por encapsular métodos em C++/CUDA diretamente nas chamadas Python da Sucuri. Nesse contexto pode-se explorar, mas não limitar-se há recuros mkFifo, troca de mensagem entre os dois modelos via Ipc.

6. Referências

Coelho, I. M.; Rios, E.; Ochi, L. S.; Boeres, C.; Farias, R; Coelho. A Benchmark on Multi Improvement Neighborhood Search Strategies in CPU/GPU Systems, WAMCA 2016.

Coelho, I. M. A Multi-Improvement GPU Strategy for Multiple Neighborhood Exploration in Combinatorial Optimization. WMC2017, Niterói

Boldt, T. Dataflow Programming Concept, Languages and Applications. INESC TEC (formerly INESC Porto) Faculty of Engineering, University of Porto Campus. (2015)

Sutherland, W.: On-Line Graphical Specification of Computer Procedures. (1966)

Ousterhout, J.: Why threads are a bad idea (for most purposes) (1996)

Wikipedia.EN. Variable neighborhood search. Disponivel em https://en.wikipedia.org/wiki/Variable_neighborhood_search#Algorithm_2:_First_improvement_.28first_descent.29_heuristic. Acessado em 08/07/2017 19:01.

Patterson, D.; Culler, D.. Slide of Introduction of Advanced Computer Architecture. Course CSCE930. Electrical Engineering and Compute SciencesUniversity of California, Berkeley. Disponivel em http://slideplayer.com/slide/5049531 e acessado em 08/07/2016 19:00.

Shen XW, Ye XC, Tan X et al. An efficient network-on-chip router for dataflow architecture. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 32(1): 11–25 Jan. 2017. DOI 10.1007/s11390-017-1703-5.

Fauzia, N.; Puchet, L-N.; Sadayappan, P. Characterizing and Enhancing Global Memory Data Coalescing on GPUs. The Ohio State University. IEEE 2015. 1-4799-8161-8/15.

Soviani, A.; Singh, J.P. OptimizingCommunicationSchedulingusingDataflowSemantics. 2009 International Conference on Parallel Processing, DOI 10.1109/ICPP.2009.66

Mittal, S.; Vetter, J.S. A Survey of CPU-GPU Heterogeneous Computing Techniques. ACM Computing Surveys, Vol. 47, No. 4, Article 69, Publication date: July 2015.

Alves, T.A.O.. Goldstein, F.; França, F. M.G.; Marzulo, L.A.J. A Minimalistic Dataflow Programming Library for Python. 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing Workshops. DOI 10.1109/SBAC-PADW.2014.20

Sena, A.C., Vaz, E.S.; França, F.M.G.; Marzulo, L. A. J., Alves, T.A.O. Graph Templates for Dataflow Programming. 2015 International Symposium on Computer Architecture and High Performance Computing Workshop. DOI 10.1109/SBAC-PADW.2015.20.+.