Przegląd narzędzi

Spis treści

1. Par4all

Par4all (PIPS) to narzędzie przeznaczone do automatycznego zrównoleglenia kodu dla naukowych programów. Narzędzie jest zbudowane z kompilatora PIPS. PIPS to kompilator typu source-to-source. PIPS powstał w 1988 r. przez firmę SILKAN. Narzędzie bazuje na technikach algebry liniowej podczas analizy. Narzędzie jest dostępne w darmowej licencji.

1. Wspierane języki programowania:
    - C,
    - Fortran,
    - CUDA,
    - OpenCL.
2. Wspierane procesory:
    - multi-core (OpenMP) processors (procesory wielordzeniowe),
    - GPGPUs (OpenCL, CUDA)
3. PIPS jest zbudowany z następujących narzędzi:
    - Newgen – służy do zarządzania strukturami danych. Zapewnia podstawowe funkcje manipulacjami strukturami danych.
    - Biblioteka Linear C3 - obsługuje wektory, macierze, wiązania liniowe (linear constraints) i wielościany oraz zapewnia podstawowe funkcje programowania liniowego. Biblioteka jest używana do analizy takich jak: testy zależności, precondition and region computation, transformacje i generowania kodu.
4. PIPS składa się z następujących elementów:
    - System make (pipsmake) – służy do zarządzania analizami i transformacjami. Jego zadaniem jest także zapewnianie spójności analiz i modułów.
    - Menadżer zasobów (pipsdbm) – służy do przechowywania wyników analiz i transformacji w bazie danych.
    - Kompilator – jest odpowiedzialny za wykonywanie analiz i transformacji podawanych przez użytkownika.
5. Kompilatory PIPS:
    - Parallelizer/vectorizer for Cray machines

- Polyhedral method (Pr. P. Feautrier)
    - Array Data Flow Graph computation.
    - Scheduling, mapping, and associated code generation.
- PUMA/WP65: Shared memory emulation
- HPFC: a HPF compiler prototype.

6. Funkcjonalności:
    - analiza międzyproceduralna,
    - generowanie kodu,
    - testowanie zależności,
    - przeprowadzanie transformacji (przekształceń),
    - używane przez narzędzia inżynierii odwrotnej,
    - zmiany w strukturach (restructurations).
7. Transformacja programu polega na pobraniu kodu, który pełni funkcję danej wejściowej, następnie kod jest modyfikowany, a wyniki przeprowadzone po modyfikacji mogą być wykorzystywane do różnych analiz. Poniżej zostały przedstawione przykładowe transformacje:
    - **Expression Optimizations** – to transformacja, która polega na optymalizacji wyrażeń za pomocą właściwości algebraicznych takich jak przemienność czy elementy neutralne. Celem transformacji jest przekształcenie wyrażeń arytmetycznych, aby zawierały mniejszą liczbę operacji.
      W celu skorzystania z tej transformacji należy użyć aliasu optimize_expressions:
      optimize_expressions > MODULE.code
      < PROGRAM.entities
      < MODULE.proper_effects
      < MODULE.cumulated_effects
      < MODULE.code
    - **Forward Substitution** – transformacja polega na zamianie wielkości skalarnych do przodu. Efektem jest cofnięcie przeprowadzonych optymalizacji, np. niezmienny ruch kodu czy wspólna eliminacja podwyrażeń.
      W celu skorzystania z tej transformacji należy użyć aliasu forward_substitute:
      forward_substitute > MODULE.code
      < PROGRAM.entities
      < MODULE.code

< MODULE.proper_effects
< MODULE.dg
< MODULE.cumulated_effects

- **Loop Distribution** – technika transformacji pętli, która pomaga optymalizacji kompilatora i dzieli pętlę na kilka pętli w tym samym zakresie indeksowania.
  Transformacja polega na rozłożeniu wszystkich pętli modułu w jak największym stopniu. W tym celu wykorzystywany jest 'Distribute Loops'. Rozkład częściowy rozprowadza instrukcję zagnieżdżonej pętli z wyjątkiem instrukcji izolowanych, które nie mają zależności na wspólnym poziomie i, są one gromadzone w tej samej i-tej pętli.
  W celu skorzystania z tej transformacji należy użyć aliasu distributer:
  distributer > MODULE.code
  < PROGRAM.entities
  < MODULE.code
  < MODULE.dg
- **Loop Interchange** – technika transformacji pętli, która polega na zamianie dwóch zmiennych iteracji stosowanych w zagnieżdżonej pętli. W przypadku pętli zagnieżdżonej pętla zewnętrzna jest zamieniana z pętlą wewnętrzną.
  Transformacja polega na pobraniu etykiety pętli i zamianie najbardziej zewnętrznej pętli z tą etykietą i najbardziej wewnętrzną pętlą w pętli zagnieżdżonej, jeśli występuje zagnieżdżona pętla.
  W celu skorzystania z tej transformacji należy użyć aliasu loop_interchange:
  loop_interchange > MODULE.code
  < PROGRAM.entities
  < MODULE.code
- **Loop Normalize** – transformacja polega na przekształceniu wszystkich pętli danego modułu w normalną formę. Norma forma jest zdefiniowana następująco: dolna granica i przyrost wynoszą 1.
  LOOP_NORMALIZE_PARALLEL_LOOPS_ONLY kontroluje, czy chcemy znormalizować tylko równoległe pętle czy wszystkie.

- **Loop Unrolling** - technika transformacji pętli, która pomaga zoptymalizować czas wykonywania programu. Zasadniczo usuwamy lub zmniejszamy iteracje. Rozwijanie pętli zwiększa prędkość programu, eliminując instrukcje kontroli pętli i instrukcje testu pętli.
  Transformacja polega na pobraniu od użytkownika: etykiety pętli w celu zlokalizowania pętli: LOOP_LABEL "" oraz współczynnika rozwijania: UNROLL_RATE.
  W celu skorzystania z tej transformacji należy użyć aliasu unroll:
  unroll > MODULE.code
  < PROGRAM.entities
  < MODULE.code
  Jeśli liczba iteracji nie jest wielokrotnością współczynnika rozwijania, to należy dodać nową pętlę i wykonać na początku lub końcu dodatkowe iteracje.
  W celu wykonania dodatkowych iteracji na końcu pętli należy ustawić LOOP_UNROLL_WITH_PROLOGUE FALSE.
  Etykiety umieszczone w ciele zostają usunięte. W przypadku rozwijania zagnieżdżonych pętli należy zacząć od najbardziej wewnętrznej pętli.
- **Partial Evaluation** – transformacja polega na wytworzeniu kodu, którego stałe liczbowe wyrażenia lub podwyrażenia są zamienione na ich wartości. Korzystanie z warunków początkowych pozwala na zamienianie niektórych stałych zmiennych typu integer na ich wartości. Transformacja nie jest wykorzystywana przy wywołaniach funkcji użytkownika.
  W celu skorzystania z tej transformacji należy użyć aliasu partial_eval:
  partial_eval > MODULE.code
  < PROGRAM.entities
  < MODULE.code
  < MODULE.proper_effects
  < MODULE.cumulated_effects
  < MODULE.preconditions
- **Strip Mining** – transformacja polega na pobraniu etykiety pętli oraz rozmiaru fragmentu lub numeru fragmenty (chunk size or number). Następnie znaleziona pętla jest dzielona na

fragmenty. W celu skorzystania z tej transformacji należy użyć aliasu strip_mine:

strip_mine > MODULE.code
< PROGRAM.entities
< MODULE.code

8. Przykładowe zmiany w strukturach:
    - Dead code elimination,
    - Cloning,
    - Atominization,
    - Declaration cleaning,
    - Control Restructuration
9. Przykładowe dane wyjściowe programów implementujących analizę międzyproceduralną:
    - Use-def chains,
    - dependence graph (graf zależności)
    - Transformers, (transformacje)
    - preconditions,
    - Symbolic complexity,
    - Effects (of instructions on data),
    - Array regions,
    - Call graph,
    - interprocedural control flow graph
10. GPU/CUDA:
    - Proces uruchamiania kerneli o dużej mocy obliczeniowej na GPU przez program sekwencyjny składa się z poszczególnych etapów:
        - przydzielenia pamięci na GPU,
        - skopiowania danych z hosta do GPU,
        - uruchomienia kernela na GPU,
        - oczekiwania hosta,
        - skopiowania wyników z GPU na hosta,
        - zwalniania pamięci na GPU.
    - Istotne wyzwania dotyczące automatycznej generacji CUDA:
        - znalezienie równoległych kerneli,
        - uzyskanie dostępu do pamięci w sposób przyjazny dla GPU,
        - zmniejszenie zużycia pamięci w GPU,

- poprawienie ponownego wykorzystania danych w kernelach, aby uzyskać lepszą wydajność, intensywność
- skorzystanie ze złożonej hierarchii pamięci

- W celu utworzenia kodu źródłowego dla GPU, należy wyodrębnić równoległy kod źródłowy do kodu źródłowego kernela.
  Poniżej został przedstawiony przykład zrównoleglenia zagnieżdżonej pętli z wykorzystaniem omp oraz dla GPU:

```
1  #pragma omp parallel for private(j)
2    for(i = 1;i <= 499; i++)
       for(j = 1;j <= 499; j++) {
4        save[i][j] = 0.25*(space[i - 1][j] + space[i + 1][j]
                          + space[i][j - 1] + space[i][j + 1]);
6      }
```

RYSUNEK 1 ZRÓWNOLEGLENIE ZAGNIEŻDŻONEJ PĘTLI ŹRÓDŁO: HTTPS://PIPS4U.ORG/PAR4ALL

```
1  p4a_kernel_launcher_0(space, save);
2  [...]
   void p4a_kernel_launcher_0(float_t space[SIZE][SIZE],
4                             float_t save[SIZE][SIZE]) {
     for(i = 1; i <= 499; i += 1)
6        for(j = 1; j <= 499; j += 1)
           p4a_kernel_0(i, j, save, space);
8  }
   [...]
10 void p4a_kernel_0(float_t space[SIZE][SIZE],
                    float_t save[SIZE][SIZE],
12                  int i,
                    int j) {
14   save[i][j] = 0.25*(space[i-1][j]+space[i+1][j]
                       +space[i][j-1]+space[i][j+1]);
16 }
```

RYSUNEK 2 ZRÓWNOLEGLENIE ZAGNIEŻDŻONEJ PĘTLI DLA GPU ŹRÓDŁO: HTTPS://PIPS4U.ORG/PAR4ALL

- Sposoby generowania komunikacji:
  - powiązanie dowolnej alokacji pamięci GPU z kopią, aby zachować jej wartość w synchronizacji z kodem hosta,
  - powiązanie dowolnego zwolnienia pamięci GPU z zachowaniem kopii kodu hosta w synchronizacji z zaktualizowanymi wartościami na GPU

W celu zapewnienia powyższych sposobów generowania komunikacji w PIPS wykorzystuje się 2 regiony:

- In-region
- Out-region

Regiony in i out można można tłumaczyć bezpośrednio za pomocą CUDA na:

- copy-in:
  cudaMemcpy (accel_address , host_address ,
  2 size , cudaMemcpyHostToDevice)
- copy-out:
  cudaMemcpy (host_address , accel_address ,
  2 size , cudaMemcpyDeviceToHost)

- Przykładowe dodanie kompilatora użytkownika:
  p4a --cuda toto.c -o toto -lm
- Infrastruktura komplikacji i linkowania:
  NVCC, NVCC+GCC, NVCC+ICC
- Obsługa GPU w PIPS opiera się na korzystaniu z wywoływania funkcji wewnętrznych z następującego pliku:
  p4a_accel-CUDA.h
- CUDA nie jest oparty na C99, ale na C89 + kilku rozszerzeniach C ++

11. Interfejsy dla użytkownika:
    - a Shell interface (Pips),
    - a line interface (Tpips),
    - an X-window interface (Wpips)
12. Instalacja PIPS:
    - z SVN: https://pips4u.org/copy_of_getting-pips/building-and-installing-pips-from%20svn
    - z SVN z autotools: https://pips4u.org/copy_of_getting-pips/building-and-installing-pips-from-svn-with-autotools
    - Instalacja PIPS jest możliwa wyłącznie na systemie Linux.

Źródła:

- https://pips4u.org/par4all
- https://pips4u.org/doc/pipsmake-rc.htdoc#x1-2150008
- https://github.com/Par4All/par4all

- https://openmpcon.org/wp-content/uploads/2018_Session4_Mosseri.pdf?fbclid=IwAR0iQ1edlm_mG6haoBp1mXsch9ChC2rpYQj_AuH9IA6ZAcb41TdokI3PozM
- https://en.wikipedia.org/wiki/Loop_unrolling
- https://en.wikipedia.org/wiki/Loop_interchange
- https://en.wikipedia.org/wiki/Loop_fission_and_fusion
- https://www.yumpu.com/en/document/read/28422154/par4all-auto-parallelizing-c-and-fortran-for-the-cuda-nvidia

2. Cetus parallel compiler

Cetus parallel compiler to narzędzie pełniące funkcje kompilatora typu source-to-source, które jest przeznaczone do transformacji programów. Cetus służy również do badań nad optymalizacjami kompilatora wielordzeniowego z wykorzystaniem automatycznej równoległości. Cetus powstał przez ParaMount Research Group na uniwersytecie Purdue w Stanach Zjednoczonych. Narzędzie jest dostępne w darmowej licencji. Narzędzie posiada interfejs graficzny.

1. Wspierane języki programowania:
    - C,
2. Wspierane procesory:
    - multi-core (OpenMP) processors (procesory wielordzeniowe),
    - GPGPUs (CUDA).
3. Wewnętrzna reprezentacja programu Cetus została zaimplementowana w Javie i została przedstawiona za pomocą hierarchii klas. Występują tam poszczególne elementy:
    - Symbol table (Tablica symboli) – jest odpowiedzialna za zapewnienie informacji na temat identyfikatorów i typów danych.
    - Traversable objects (Obiekty Traversable) – wszystkie obiekty wewnętrznej reprezentacji programu, które pochodzą z klasy Traversable. Klasa Traversable jest odpowiedzialna za iterowania list obiektów.
    - Iterators (Iteratory) - BreadthFirst, DepthFirst i Flat iterators to wbudowane iteratory, które służą do łatwego wyszukiwania wewnętrznej reprezentacji programu.
    - Annotations (Adnotacje) - komentarze, pragmy, dyrektywy i inne rodzaje pomocniczych informacji o obiektach wewnętrznej reprezentacji programu.
    - Printing – funkcje drukowania, które służą do renderowania klas wewnętrznej reprezentacji programu.
4. Funkcjonalności:
    - analiza programu,
    - przeprowadzanie transformacji,
5. Przykładowe analizy:

- **Symbolic manipulation –** to analiza programu w formie warunków symbolicznych. Manipulacja wyrażeniami symbolicznymi jest bardzo ważna, gdy projektowanie algorytmów analizy dotyczy programów rzeczywistych. Cetus wspiera manipulacje wyrażeniami z narzędziami, które upraszczają i normalizują wyrażenia symboliczne.
- **Array section analysis –** array sections służą do opisu zestawów elementów tablicy dostępnych w instrukcjach programu. Analiza polega na wykonaniu analizy zmiennych tablicowych dla danych wejściowych programu poprzez obliczenie zakresu wartości indeksów tablicy. Jeśli występuje wiele instrukcji, to są one scalane.
- **Data dependence analysis** – analiza polega na identyfikacji referencji danych, które uzyskują dostęp do tej samej lokalizacji pamięci podczas wykonywania programu oraz występują zależności między tymi referencjami. Przykładową analizą zależności danych jest analiza zależności danych tablicowych. Analiza zależności danych tablicowych polega na analizie indeksów tablicy.
- **Range analysis –** analiza polega na obliczaniu zakresów wartości zmiennych całkowitych w każdym punkcie programu. Po dokonanej analizie zwracana jest mapa z każdej instrukcji do zestawu zakresu wartości ważnych przed każdą instrukcją.

6. Przykładowe transformacje:
    - **Privatization –** transformacja polega na identyfikacji prywatnych zmiennych w pętli. Prywatna zmienna pełni funkcję zmiennej tymczasowej w pętli, która jest zapisywana jako pierwsza i później używana w pętli. Array sections zapewniają tymczasowe lokalizacje dla prywatnych zmiennych tablic. Cetus zawiera implementację array privatizer, który służy do obsługi array sections zawierających warunki symboliczne.
    - **Reduction variable recognition** – transformacja polega na wykryciu operacji redukujących, które są istotne przy automatycznym zrównolegleniu wielu pętli. Podstawowymi kryteriami przy szukaniu operacji redukujących są:

    - pętla zawiera jedno lub kilka wyrażeń przypisania w następującej postaci: rv = rv + expr, gdzie rv to zmienna skalarna lub tablica dostępu, expr to wyrażenie pętli o wartości rzeczywistej,
    - rv nie występuje nigdzie indziej w pętli.
  - **Induction variable substitution –** transformacja polega na rozpoznawaniu i podstawianiu zmiennych indukcyjnych. Instrukcja indukcyjna ma podobną postać do operacji redukującej: rv=rv+expr i musi być zastąpiona przez inną postać, która nie powoduje zależności między danymi.
7. GPU/CUDA:
  - CETUS zawiera automatyczny translator z OpenMP do CUDA GPU, który umożliwia konwertowanie równoległości w OpenMP, w szczególności równoległości na poziomie pętli ze względu na występujące podobieństwa pomiędzy modelami programowania OpenMP a CUDA. Różnice między architekturą OpenMP i GPGPU przyczyniły się do powstania różnic w technikach optymalizacji. W celu optymalizacji wykonywania programów w CUDA zostały opracowane techniki transformacji. Zostały wykorzystane następujące techniki transformacji:
    - loop interchange,
    - loop coalescing.
8. Narzędzia potrzebne do instalacji:
  - JAVA 2 SDK, SE 1.5.x (lub późniejsza),
  - ANTLRv2,
  - GCC
9. Instalacja:
  - https://engineering.purdue.edu/Cetus/Documentation/release_notes/release-notes-1.4.4.txt
  - https://engineering.purdue.edu/Cetus/

Źródła:

- https://engineering.purdue.edu/Cetus/
- https://engineering.purdue.edu/paramnt/publications/ieeecomputer-Cetus-09.pdf
- https://engineering.purdue.edu/paramnt/publications/CPC09.pdf
- https://engineering.purdue.edu/Cetus/Documentation/tutorials/ppopp09.pdf

3. PPCG

PPCG (Polyhedral Parallel Code Generator) to narzędzie pełniące funkcje kompilatora typu source-to-source, które jest oparte na wielościennych technikach kompilacji. Narzędzie łączy transformacje afiniczne w celu wyodrębnienia równoległości danych z generatorem kodów.

1. Wspierane języki programowania:
    - C,
2. Wspierane procesory:
    - multi-core (OpenMP) processors (procesory wielordzeniowe),
    - GPGPUs (CUDA),
3. Kompilatory, które wykorzystują struktury wielościenne do wykonywania transformacji pętli:
    - PoCC,
    - Pluto,
    - CHiLL,
4. Kompilatory, które wykorzystują struktury wielościenne do kompilacji dla architektur wielordzeniowych:
    - GCC,
    - LLVM,
    - IBM XL,
5. Funkcjonalności:
    - analiza zależności,
    - generowanie kodu,
    - wykonywanie transformacji afinicznych,
    - ekstrakcja modelu,
    - wykonywanie wielopoziomowej strategii kafelkowania.
6. **Multilevel tiling strategy** (wielopoziomowa strategia kafelkowania) – strategia jest dostosowana do wielu poziomów równoległości i hierarchii pamięci akceleratorów GPU. Polega na oddzieleniu wielopoziomowej równoległości od optymalizacji lokalizacji. Umożliwia wybieranie liczby wątków i bloków niezależnie od bloków przypisanych do rejestrów i pamięci współdzielonej,
7. **Model extraction** (ekstrakcja modelu) – polega na utworzeniu modelu wielościennego składającego się z domeny iteracji, relacji

dostępu oraz harmonogramu na podstawie pobranego kodu w języku programowania C jako danej wejściowej. Ekstrakcja modelu jest wykonywana za pomocą pet.

8. **Dependence analysis** (analiza zależności) – polega na określeniu zależności pomiędzy instrukcjami iteracyjnymi na podstawie następujących danych wejściowych: domeny iteracji, relacji dostępu oraz harmonogramu. Zależności w programie można przedstawić za pomocą relacji zależności. Analiza zależności jest wykonywana za pomocą isl.
9. **Scheduling** – polega na utworzeniu nowego harmonogramu. Tworzenie nowego harmonogramu składa się z następujących etapów:
    - Ujawnienie równoległości i kafelkowanie – jest wykonywane za pomocą isl, który bazuje na algorytmie „Pluto" w celu wykorzystania wielu poziomów równoległości i hierarchii pamięci akceleratorów GPU. Używane jest afiniczne harmonogramowanie, które polega na przekształceniu zagnieżdżonej pętli w pętli opartej na kafelkowaniu przy odpowiednio wybranej wielowymiarowej funkcji afinicznej.
    - Mapowanie na host i GPU – polega na decydowaniu, która część harmonogramu ma być wykonywana na hoście, a która na GPU.
    - Kafelkowanie i mapowanie do bloków i wątków – polega na wykorzystaniu zrównoleglenia i kafelkowania. W każdym kernelu odbywa się proces kafelkowania na podstawie rozmiaru określonego przez użytkownika. Kafelkowanie służy do dzielenia równoległych pętli na kilka pętli. Dwie zewnętrzne równoległe pętle oparte na kafelkowaniu są mapowane do bloków, a trzy wewnętrzne równoległe pętle są mapowane do wątków w bloku.
10. Generowanie kodu – polega na wygenerowaniu kodu w języku programowania C lub CUDA, który odwiedza każdy element w domenie iteracji w kolejności zgodnej z podanym harmonogramem. W PPCG wykorzystywany jest własny generator kodu zaimplementowany w isl.
11. Transformacje afiniczne:
    - Transformacje afiniczne są wykorzystywane w celu odciążenia obliczeń równoległych do GPU oraz

wyodrębnienia równoległości danych z generatorem kodów. Za pomocą transformacji afinicznych wyrażane są procesy. Przy ekstrakcji modelu każdy element: domena iteracji, relacja dostępu oraz harmonogram są zapisywane za pomocą ograniczeń afinicznych. Ograniczenia afiniczne mogą być wykorzystywane przy kafelkowaniu.

12. GPU/CUDA:
    - Zarządzanie pamięcią – składa się z kilku etapów:
        - Transfer danych na GPU i na host – to pierwszy etap zarządzania pamięcią, który polega na przesłaniu danych do GPU. W tym celu należy przydzielić miejsce dla dostępnych tablic na urządzeniu oraz skopiować dane do i z urządzenia, ponieważ przestrzeń pamięci urządzenia host jest oddzielona od przestrzeni pamięci urządzenia CUDA. Każda dostępna tablica jest alokowana w całości na urządzeniu. Tablice, których elementy są używane bez wcześniejszego zdefiniowania na wejściu programu, są kopiowane na urządzenie. Tablice, których elementy zostały zaktualizowane są kopiowane na urządzenie po zakończeniu wszystkich kerneli. W przypadku skalarów, które zostały zaktualizowane przez kernel, są traktowane jako tablice zero-wymiarowe. Skalary, które mają być wykorzystywane tylko do odczytu, są przekazywane jako argumenty do kernela.
        - Detekcja grup odwołań/ referencji do tablicy – to drugi etap zarządzania pamięcią, który polega na grupowaniu odwołań do tablicy, które pojawiają się w kodzie wejściowym w celu skopiowania części pamięci globalnej do pamięci współdzielonej lub do rejestrów.
        - Alokacja do rejestrów i pamięci współdzielonej – to trzeci etap zarządzania pamięcią, który polega na wybraniu elementów tablicy dostępnych za pomocą wybranej grupy dla każdej iteracji pętli opartej o kafelkowanie. Znalezione elementy, które można skopiować do pamięci współdzielonej, stanowią relację pomiędzy iteratorami pętli opartej o kafelkowanie wraz ze wszystkimi iteratorami pętli zewnętrznej

mapowanymi na hoście a tablicą wskaźników. Do mapowania do rejestrów należy uwzględnić relację między punktami pętli, które zostaną zmapowane na wątki a elementami tablicy w pętli zewnętrznej. Decyzja o miejscu, gdzie mają zostać umieszczone dane jest podejmowana na podstawie obliczonych wartości kafelków oraz czy zostaną one ponownie użyte.

- Zagnieżdżone pętle są dzielone na wiele kernelów GPU. Każda instrukcja obliczeniowa może być zmapowana na GPU, pozostawiając tylko kod sterujący na procesorze hosta.

13. Instalacja:
    - https://github.com/Meinersbur/ppcg

Źródła:

- https://www.researchgate.net/publication/256121128_Polyhedral_Parallel_Code_Generation_for_CUDA
- https://dl.acm.org/doi/pdf/10.1145/2400682.2400713