Экспериментальное исследование специализатора GPGPU-программ AnyDSL

Автор: Тюрин Алексей Валерьевич, группа 16.Б11-мм

Научный руководитель: к.ф-м.н., доцент С. В. Григорьев

Научный консультант: к.ф-м.н. Д. А. Березун

Рецензент: стажёр-исследователь «ИСП им. В. П. Иванникова РАН» Е. Ю. Шарыгин

Санкт-Петербургский государственный университет
Кафедра системного программирования

13 июня 2020г.

Графические процессоры

- Повсеместно используются для ускорения вычислений
  - CUDA
- Задачи выполняются параллельно большим числом потоков
  - Функция, выполняемая на GPU, называется *ядром*
- Операции доступа к памяти часто превалируют над всеми остальными операциями
  - Область для различного рода оптимизаций

Организация памяти

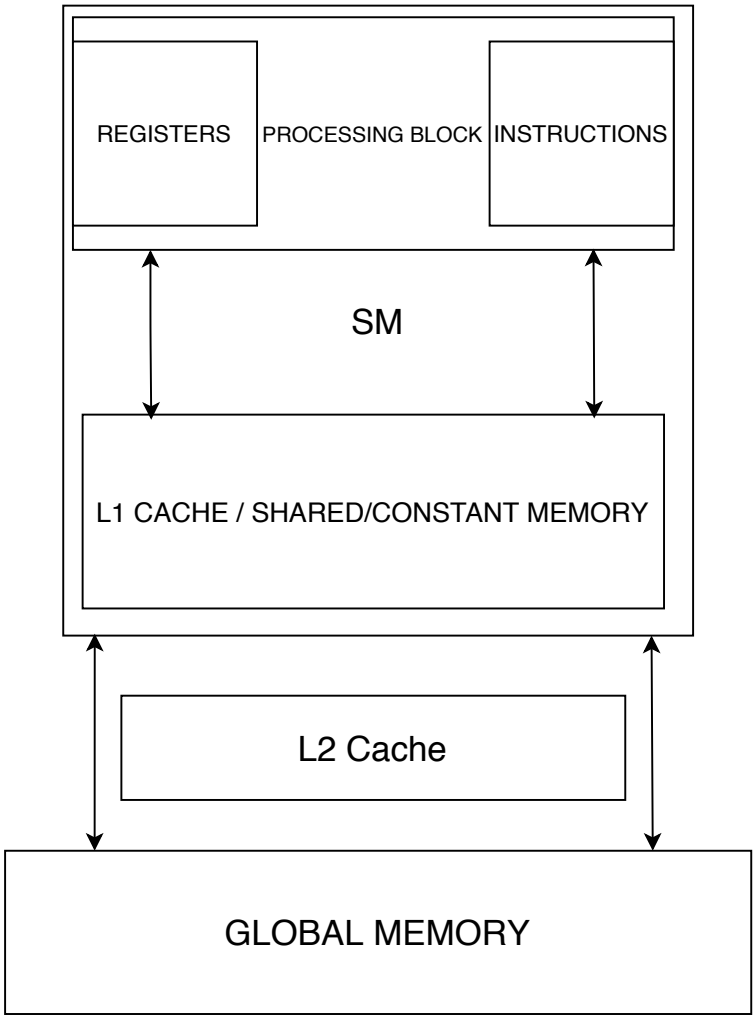


Рис.: Иерархия памяти видеокарты

- Различный объем
- Различная скорость доступа
- Ручное управление
- Оптимизации
  - Пул памяти
  - Распределители памяти
  - Перемещение регистров в разделяемую память
  - Автоматическое выделение
  - Не учитывают статичность параметров
- Можно предложить подход, использующий специализацию

Частичные вычисления или специализация

$$[\![\underbrace{handleData}_{handleData}]\!][filterParams, data] = \underbrace{[\![\overbrace{mix}^{\text{partial evaluator}}]\!][handleData, filterParams]}_{handleData_{mix}}$$

```
handleData (filterParams, data)
{
  res = new List()
  for d in data
     for e in filterParams
        if d % e == 0
        then res.Add(d)
  return res
}
```

Частичные вычисления или специализация

$$[\![\underbrace{handleData}_{handleData}]\!][filterParams, data] = \underbrace{[\![\overbrace{mix}^{\text{partial evaluator}}]\!][handleData, filterParams]}_{handleData_{mix}}$$

$$[\![[\![mix]\!][handleData, [2; 3]]\!]$$

```
handleData (filterParams, data)
{
  res = new List()
  for d in data
     for e in filterParams
        if d % e == 0
        then res.Add(d)
  return res
}
```

```
handleData (data)
{
  res = new List()
  for d in data
    if d % 2 == 0 ||
       d % 3 == 0
    then res.Add(d)
  return res
}
```

Частичные вычисления или специализация

$$[\![\underbrace{handleData}_{handleData}]\!][filterParams, data] = \underbrace{[\![\overbrace{mix}^{\text{partial evaluator}}]\!][handleData, filterParams]}_{handleData_{mix}}$$

$$[\![[\![mix]\!][handleData, [2; 3]]\!]$$

```
handleData (filterParams, data)
{
  res = new List()
  for d in data
     for e in filterParams
        if d % e == 0
        then res.Add(d)
  return res
}
```

```
handleData (data)
{
  res = new List()
  for d in data
     if d % 2 == 0 ||
        d % 3 == 0
     then res.Add(d)
  return res
}
```

Частичные вычисления или специализация

```
for d in data
 for e in filterParams
    if d % e == 0
    then res.Add(d)
```

```
for d in data
 if d % 2 == 0 ||
    d % 3 == 0
 then res.Add(d)
```

- Встраивание данных в код может уменьшить число транзакций в память
- Части программы, зависящие только от встроенных данных можно предвычислить
- Специализация во время исполнения
- Накладные расходы на специализацию можно спрятать

Постановка задачи

- Цель
  - Экспериментально исследовать возможность оптимизации GPGPU-программ с помощью специализации доступа к статическим данным, используя фреймворк AnyDSL

- Задачи
  - Реализовать экспериментальные сценарии для GPU на CUDA и AnyDSL
  - Собрать данные для экспериментов
  - Провести эксперименты и проанализировать результаты

Подходы

- JIT-компиляция ядра
- Специализаторы
  - Специализация непосредственно GPU ядер слабо изучена
  - AnyDSL
    - ★ Онлайн-специализатор
    - ★ Работает поверх своего внутреннего представления и *Impala* DSL
    - ★ Генерирует CUDA код
    - ★ Компонуется с C++
    - ★ В ядре нельзя вызывать определенные в других файлах функции
  - LLVM.mix
    - ★ Оффлайн-специализатор
    - ★ Работает поверх LLVM IR
    - ★ В настоящее время при работе с CUDA возникают конфликты
- Суперкомпиляторы
  - Для функциональных языков
  - Из-за обилия конструкторов производят огромный код даже на игрушечных примерах

Архитектура

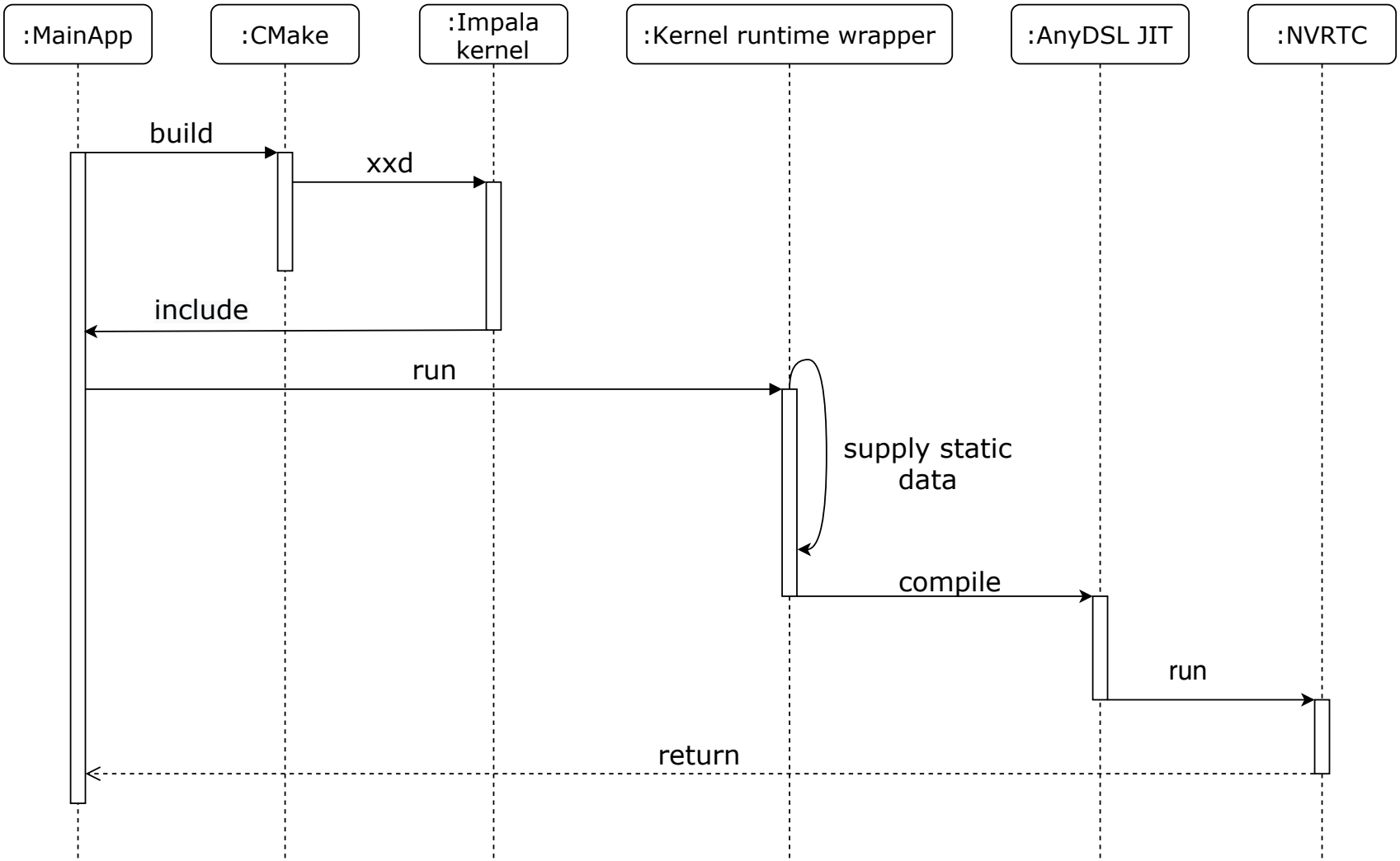


Рис.: Структура решения

Эксперименты

- Сценарии
  - Наивный поиск подстроки в строке
  - Наивный поиск нескольких подстрок в строке
  - Ахо Корасик
  - Сверточный 2-D фильтр
- Подход
  - Версии на AnyDSL и CUDA C максимально похожи
  - Видеокарта NVIDIA Tesla T4
  - Измеряется время исполнения ядра на видеокарте
  - Константная память

Наивный поиск подстроки

- КМП-тест
- Целевая строка: случайный набор символов
- Шаблоны: случайные шаблоны длины 12

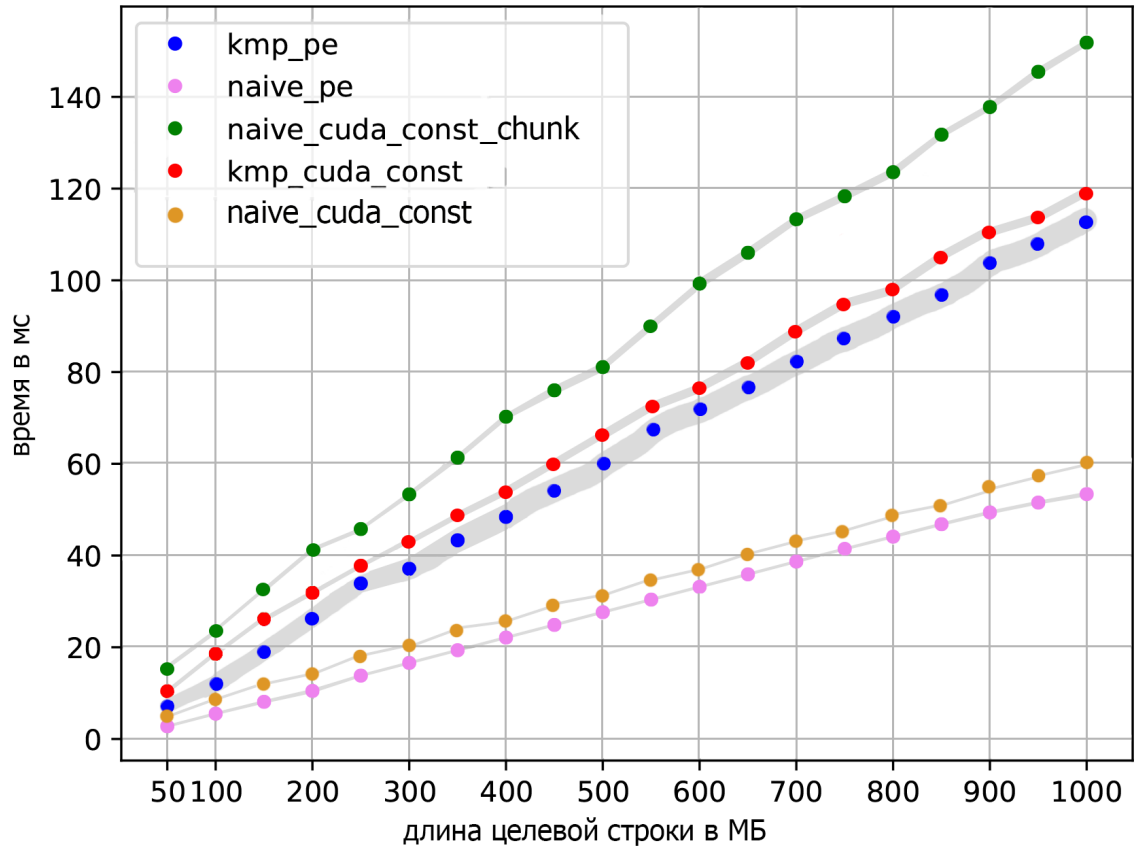


Рис.: Наивный поиск подстроки

Наивный поиск подстроки

$$[\underbrace{[matcher]}_{matcher}][pattern, string] = [\underbrace{[[mix][matcher, pattern]]}_{matcher_{mix}}][string]$$

```
1    LDG.E.U8 R8, [R8] ;//loads value from global memory into
     register
2    LDC R6, c[0x3][R6] ; // loads another value from constant
    memory into register
3    BFE R13, R8, 0x1000 ;
4    BFE R12, R6, 0x1000 ;
5    ISETP.NE.AND P0, PT, R13, R12, PT ; //compare pre-
    loaded registers
```

Листинг 1: matcher

Наивный поиск подстроки

$$[\![\underbrace{matcher}_{matcher}]\![patterns, string] = [\![\underbrace{[\![mix]\![matcher, patterns]}_{matcher_{mix}}]\![string]$$

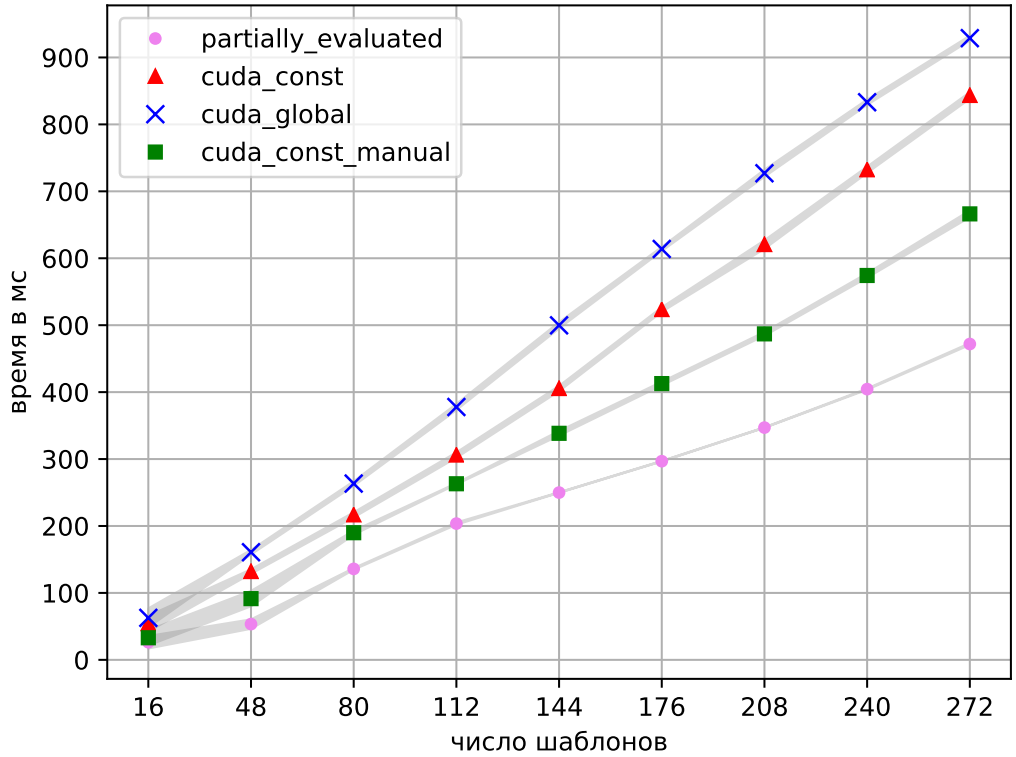
```
1     LDG.E.U8 R0, [R4] ; //loads value from global memory
      into register
2     BFE R6, R0, 0x1000 ;
3     ISETP.NE.AND P0, PT, R6, 0x49, PT ; //Put 0x49 right
      into a compare instruction parameter
```

Листинг 2: matcher_{mix}

- Доступ к встроенным данным осуществляется через кэш инструкций

Наивный поиск нескольких подстрок

- Целевая строка: Botnet трафик[1] в 500 МБ
- Подстроки: шаблоны из Snort V3[2]

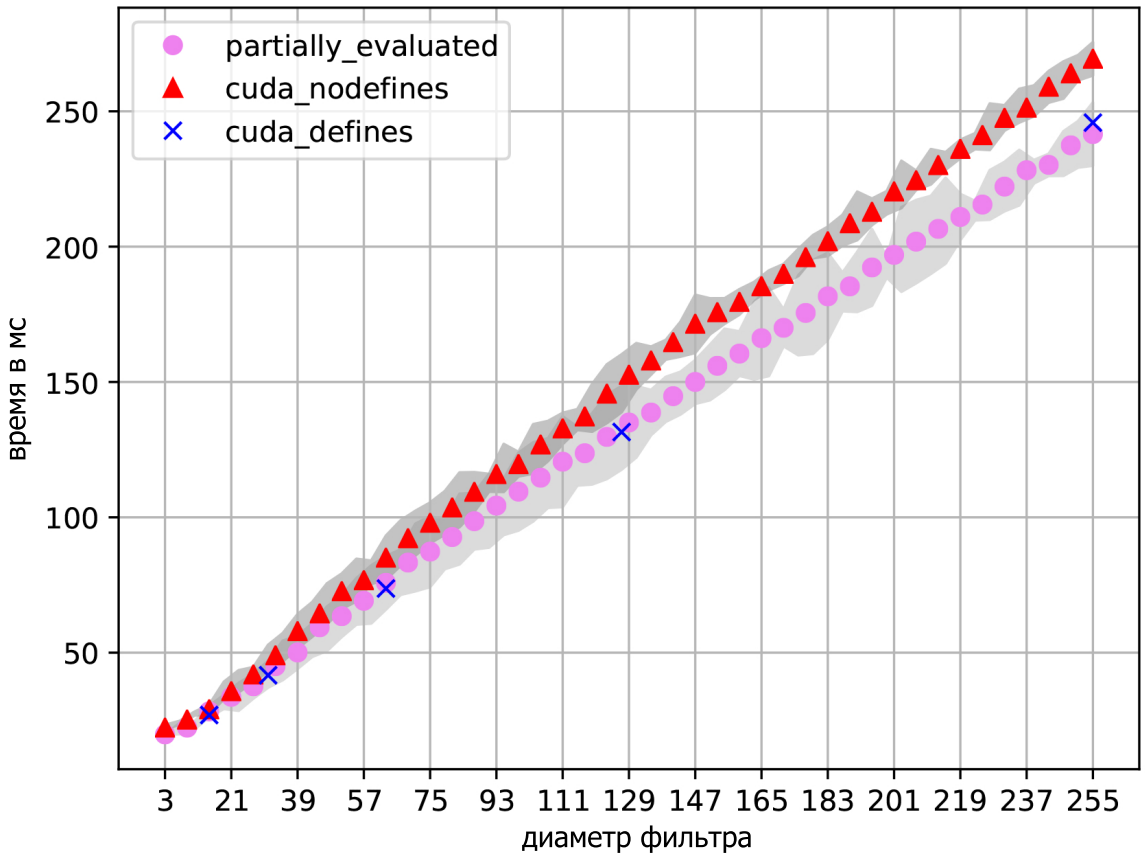


[1] https://www.unb.ca/cic/datasets/botnet.html

[2] https://www.snort.org/downloads

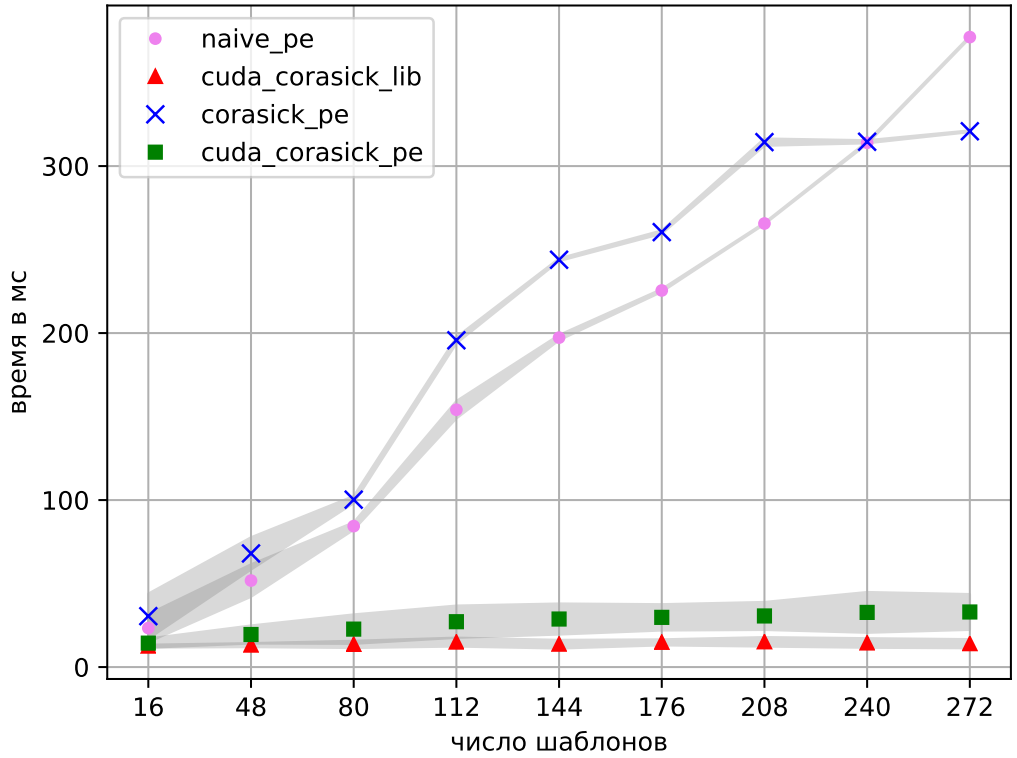
Сверточный фильтр

- Картинка: случайный 2-D массив в 1 GB
- Фильтры: случайные квадратные фильтры размером до 255x255



Ахо-Корасик[3]

- Целевая строка: Botnet трафик в 500 МБ
- Подстроки: шаблоны из Snort V3



[3] http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0054-GTC2012-PFAC-GPU-Algorithm.pdf

Результаты

- ✔ Реализованы[4] экспериментальные сценарии
- ✔ Проведены эксперименты
- ✔ Проанализированы результаты
  - Специализация доступа к памяти в некоторых случаях применима для оптимизации GPGPU-программ
    - ★ Показывает хорошие результаты, когда удается сократить количество доступов к памяти
    - ★ Доступ к встроенным данным осуществляется через кэш инструкций.
    - ★ При этом управление памятью более автоматизировано
    - ★ Может генерироваться более производительный код
    - ★ Расхождение потоков при исполнении плохо влияет на производительность при попытках встраивания данных, доступ к которым осуществляется по динамическим индексам
    - ★ Результат специфичен для конкретного компилятора и устройства
- ✔ Часть результатов опубликована[5] как постер на *PPOPP-2020*

[4] https://github.com/Tiltedprogrammer/spec

[5] https://dl.acm.org/doi/abs/10.1145/3332466.3374507