



---

# Система генерации GPU-кода для модели COSMO

---

## ОТЧЁТ ПО ЭТАПУ I

Д. Н. Микушин  
dmikushin@nvidia.com

А. А. Мыльцев  
alexander.myltsev@gmail.com

А. П. Петров  
artem.petrov80@gmail.com

Ю. В. Мартынова  
foxyj13@gmail.com

М. Штемберг  
warchild1000@mail.ru

23 июля 2011 г.

# Оглавление

<b>1</b>	<b>Краткое описание проекта</b>	<b>1</b>
1.1	Цель работы . . . . .	1
1.2	Метод . . . . .	1
1.3	Этапы работы . . . . .	2
1.4	Основные результаты первого этапа . . . . .	3
<b>2</b>	<b>Основные сведения о модели COSMO</b>	<b>5</b>
<b>3</b>	<b>Выбор оптимальной программной модели</b>	<b>7</b>
3.1	PGI CUDA Fortran . . . . .	7
3.2	PGI Accelerator . . . . .	8
3.3	PathScale HMPP / OpenHMPP . . . . .	9
3.4	f2c-acc . . . . .	9
3.5	Специализированный язык шаблонов SCS . . . . .	11
3.6	par4all . . . . .	12
3.7	Обоснование выбора программной модели . . . . .	14
<b>4</b>	<b>Система генерации кода для GPU</b>	<b>16</b>
4.1	Трансформация исходного кода . . . . .	16
4.2	CUDA-backend на основе LLVM . . . . .	18
4.3	Runtime-библиотека . . . . .	18
4.4	Общая схема генератора . . . . .	19
<b>5</b>	<b>Генерация GPU-кода для модели COSMO</b>	<b>24</b>

<b>6</b>	<b>Руководство пользователя</b>	<b>25</b>
6.1	Опции компилятора . . . . .	25
6.2	Настройка среды . . . . .	26
6.3	Компиляция COSMO . . . . .	27
6.4	Компиляция WRF . . . . .	28
6.5	Ограничения . . . . .	29
<b>7</b>	<b>Дополнительные сведения для разработчиков</b>	<b>30</b>
7.1	Компиляция компонентов системы . . . . .	30
7.1.1	LLVM . . . . .	30
7.1.2	GNU GCC 4.5 . . . . .	31
7.2	DragonEgg . . . . .	32
<b>8</b>	<b>Лицензионный статус</b>	<b>33</b>
<b>9</b>	<b>План развития</b>	<b>34</b>
9.1	Первый этап (окончен) . . . . .	34
9.2	Второй этап . . . . .	35
9.2.1	Задачи поддержки/совместимости . . . . .	35
9.2.2	Задачи производительности/эффективности . . . . .	36
9.3	Предложения координационного характера . . . . .	37

## **Аннотация**

Работа представляет собой описание совместной деятельности СибНИГМИ и NVIDIA по созданию и внедрению технологий использования GPU в оперативных моделях прогноза погоды. По материалам приоритетного проекта РОМРА дана характеристика вычислительных аспектов модели COSMO. На основе анализа других работ последовательно выстраивается аргументация для создания системы преобразования исходного кода с оптимальными свойствами. Реализуемый метод генерации GPU-ядер без изменения исходного кода модели по-видимому является наиболее мягким и консервативным из всех возможных решений с точки зрения пользователя и наиболее трудоёмким с точки зрения разработчика. Приведены результаты первого этапа работы, обзор похожих решений, детали реализации, краткое руководство пользователя и разработчика, предложен план дальнейшего развития системы.

# Глава 1

## Краткое описание проекта

### 1.1 Цель работы

Цель данной работы состоит в поэтапной разработке метода использования GPU в коде численных моделей на языке Fortran, настроенного на наилучшую поддержку используемых моделей. Предполагается, что метод может занять в настоящее время пустующую нишу средства обеспечения максимально плавного перехода к использованию GPU в программах для языка Fortran. Для этого в первую очередь требуется, чтобы исходный код пользователя по умолчанию компилировался для GPU **в оригинальном виде**, а необходимость учета факторов, требующих от пользователя специфических знаний о программировании GPU возникала **только по его желанию**. Важно обеспечить достаточную эффективность расчетов на GPU и высокое качество результатов, что имеет особую важность для оперативных моделей прогноза погоды.

### 1.2 Метод

Для использования GPU в модели COSMO применен метод автоматической генерации GPU-кода на базе готовых технологий GNU Compiler Toolchain, XSLT, LLVM и NVIDIA CUDA Toolkit. Специально для этой задачи разработан метод автоматического расщепления исходного кода на языке Fortran на части для исполнения на GPU

и CPU, включающий транслятор вида Fortran95  $\rightarrow$  Fortran95  $\rightarrow$  Fortran95 + CUDA C и runtime-библиотеку. Библиотека реализует функции запуска ядер и синхронизации данных между GPU и CPU с учетом косвенных зависимостей, таких как данные в модулях. Функции также реализуют специальный режим контрольного расчета, при котором все вычисления производятся одновременно на CPU и GPU с проверкой на условие предельного уровня расхождения результатов для каждого отдельного ядра.

## 1.3 Этапы работы

Цель проекта предполагается достигнуть в два этапа:

1. реализация общей схемы, тестирование, проверка правильности результатов модели COSMO (апрель-июнь)
2. вычислительная эффективность, повышение доли параллельных циклов, использующих GPU (июль-сентябрь)

На первом этапе должна быть подготовлена и реализована структурная схема компилятора, состоящая как из готовых компонентов, так и специально разработанных. Программная оболочка уже должна иметь достаточно завершённый вид для применения конечным пользователем. Необходимо представить убедительные результаты надёжной и корректной работы метода в сложных приложениях, что делает оправданным работы по его дальнейшему улучшению и внедрению.

Приоритеты второго этапа — доля портированных на GPU циклов, производительность вычислительного кода на GPU, эффективность обмена данными. Если результатом первого этапа является надёжно работающая модель, то результат второго — работающая надёжно и быстро. Предполагается, что основные работы могут быть связаны с *выпуклым анализом* (polyhedral analysis) вычислительных циклов, оптимальным позиционированием обменов данными в соответствии с графом зависимостей, использованием shared-памяти в GPU-ядрах.

## 1.4 Основные результаты первого этапа

Компилятор с генерацией GPU-ядер реализован и может быть использован конечным пользователем. Проведено тестирование модели COSMO в режиме сравнения с контрольными результатами, для подавляющего числа запусков GPU-ядер получены расхождения результатов менее установленного порога  $10^{-4}$  для вещественных массивов и точное совпадение результатов для целочисленных величин (рис. 1.1).

# 20 steps of COSMO Model with 54 auto-generated loops running on GPU

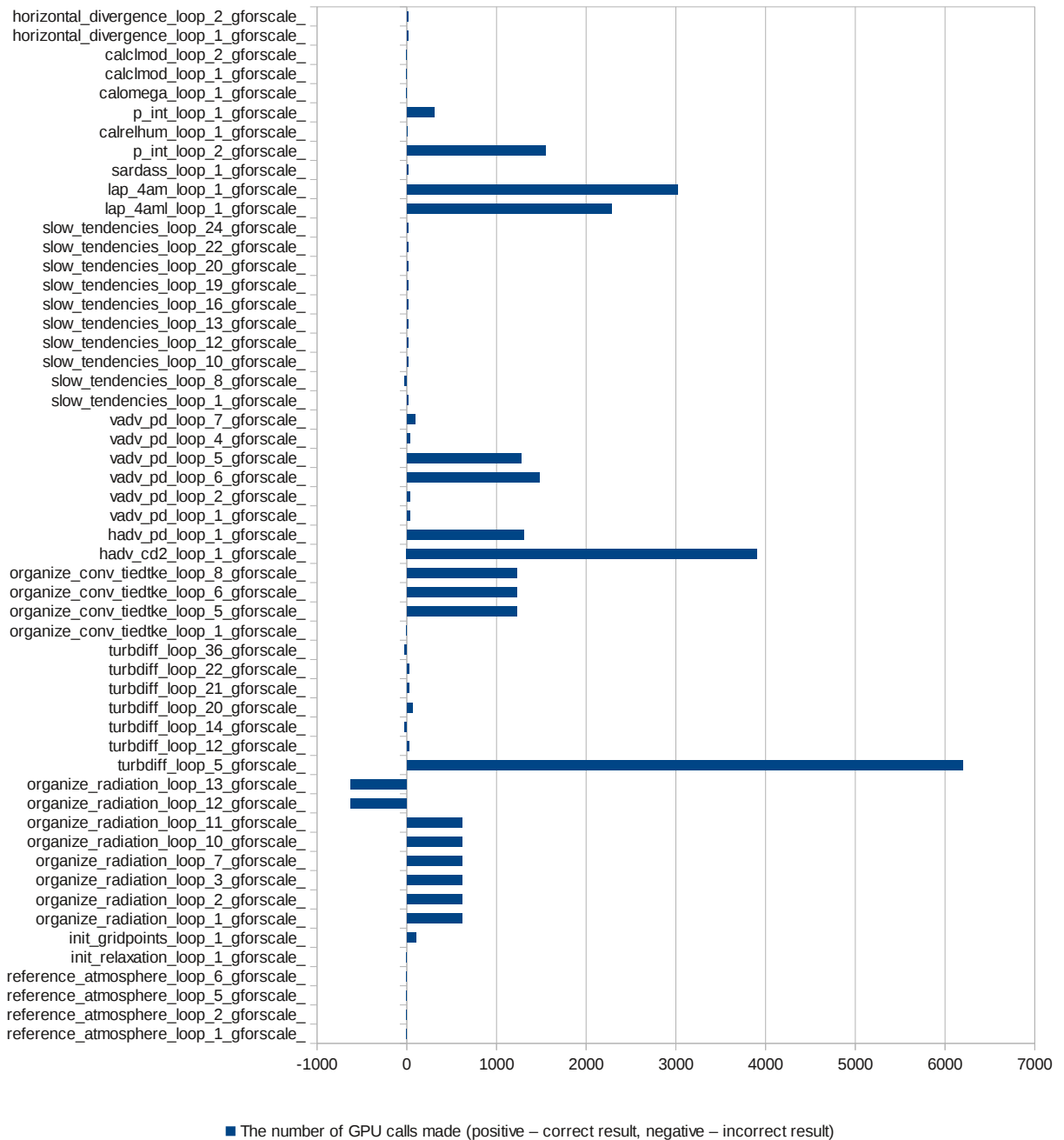


Рис. 1.1: Тестовый запуск модели COSMO с 54 сгенерированными GPU-ядрами в режиме сравнения результатов



## Глава 2

# Основные сведения о модели COSMO

Модель COSMO — региональная разностная модель атмосферы и подстилающей поверхности, используемая в качестве оперативной модели метеослужбами ряда стран Европы и Гидрометцентром РФ. Вычислительное ядро представлено единым приложением с исходным кодом на языке Fortran, около 250 тыс. строк. Распределение вычислений реализуется при помощи двумерной декомпозиции горизонтальной широтно-долготной сетки и интерфейса MPI. Производительность модели составляет 3-13% пиковой в зависимости от используемой вычислительной системы. Такое значение типично для моделей подобного класса и может быть обусловлено, главным образом, пропускной способностью памяти и коммуникационной сети (рис. 2.1).

Разработчики модели отмечают привлекательные характеристики GPU — энергоэффективность, большая пропускная способность памяти, меньшая цена. С другой стороны, развитие GPU-реализации сдерживается рядом факторов — сложностью, затратами на поддержку обратной совместимости и малой выгодой от частичного портирования кода (рис. 2.2). В числе недостатков также отмечена необходимость привязки к закрытой технологии конкретного производителя (рис. 3.5).

## Current Situation of the COSMO-Model

OR: Why has the COSMO-Model problems with new architectures?

- It only uses a pure MPI parallelization. Within one MPI task it is written for a flat memory access.
  - NEC SX-9: 13 % of peak
  - IBM pwr6: about 5-6 % of peak
  - Cray XT4: about 3-4 % of peak (?)
- One part is becoming more and more serious: I/O
  - many global communications involved
  - disk access is purely sequential
- Although the communications besides I/O are almost all local, the speedup degrades when using many processors.

03.04.05.11

POMPA Kick-Off Meeting



7

Рис. 2.1: Проблемы вычислительной эффективности модели COSMO (Ulrich Schättler)



## GPU Implementation

- **GPUs are very attractive...**
  - higher memory bandwidth
  - higher FLOPS per Watt
  - low cost
- **But...**
  - need to consider the whole workflow
  - COSMO is a community code and will always (?) have to run on a commodity cluster
  - the code should stay easy to understand and modify
- New project proposal (OPCODE) for a GPU implementation of MeteoSwiss operational suite submitted

Рис. 2.2: Сочетание преимуществ GPU и требований к модели (Oliver Fuhrer)

## Глава 3

# Выбор оптимальной программной модели

Конкретным результатом данной работы должна стать **востребованная** GPU-версия модели COSMO. Обеспечение востребованности является намного более сложной задачей, чем простое портирование кода на GPU. Например, для LINPACK фактически достаточно получить правильный результат решения стандартной задачи быстрее конкурирующих систем. При этом не так уж и важно, каким образом он достигнут и насколько сложно будет сделать то же самое другим разработчикам по отношению к своим задачам, если они имеют меньше знаний о вычислительной специфике. Работа над COSMO — это противоположный случай: необходимо учесть интересы пользователей и разработчиков модели, с тем чтобы обеспечить дальнейшее развитие GPU-реализации силами сообщества. Поэтому ориентиром для работы являются идеи, развиваемые в рамках приоритетного проекта РОМРА. При выборе направления работы проанализирован ряд перспективных подходов, представленных на РОМРА Kick-off Meeting, окончательное решение основано на анализе их преимуществ и недостатков.

### 3.1 PGI CUDA Fortran

ФФ



### GPU implementation using PGI directives

- Kernel generated at compilation time by adding directives in the code.  
Example of a matrix multiply to be compiled for an accelerator

```
!$acc region
do k = 1,n1
  do i = 1,n3
    c(i,k) = 0.0
    do j = 1,n2
      c(i,k) = c(i,k) + a(i,j) * b(j,k)
    enddo
  enddo
enddo
!$acc end region
```

- Grid and block sizes are automatically set by the compiler or can be manually tuned using the **parallel** and **vector** keywords
- **Mirror** and **reflected** keywords enable to declare GPU resident data arrays, thus avoiding data transfer between multiple kernel calls.
- Based on other codes experiences (WRF, fluid dynamics) <sup>1</sup>, directly adding directives to existing code may not be very efficient : still requires some re-writing to get better performance (loop reordering ...)
- Limitations : calls to subroutines within acc region need to be inlined, ...

<sup>1</sup> <http://www.pgroup.com/resources/accel.htm>

Рис. 3.1: Особенности программирования директивами PGI Accelerator (Xavier Lapillonne, Oliver Fuhrer)

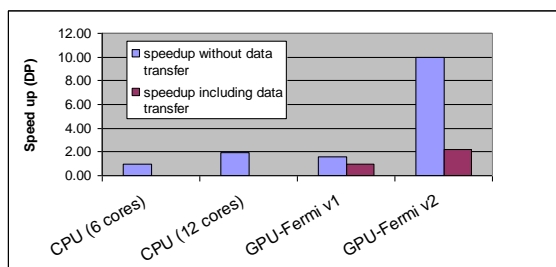
## 3.2 PGI Accelerator

Программная модель PGI Accelerator является системой разметки параллельных циклов, встроенной в язык, подобно OpenMP. В директивах PGI Accelerator указываются зависимые данные вычислительного цикла и, в случае необходимости, режим их пересылки между памятью CPU и GPU (рис. 3.1). Для GPU-ядра, полученного при компиляции цикла с директивой `$acc region`, компилятор автоматически выбирает количество рабочих блоков и потоков.

Для блока облачной микрофизики с помощью PGI Accelerator и GPU архитектуры Fermi продемонстрирована возможность получения двукратного прироста производительности по сравнению с 12-ядерным процессором (рис. 3.2). В то же время авторы отмечают большой объём необходимой ручной работы по оптимизации кода, например, по перестановке циклов (впрочем из презентации не ясно, была ли проведена аналогичная перестановка в коде для CPU и не приводит ли она к аналогичному приросту производительности), рис. 3.3.



### Using Fermi card and double precision real



- 10x speed up achieved with Fermi card and code version 2.
- Based on theoretical peak performance from 6 cores Opteron and Fermi card, one would expect a 9x speed up.
- Only 2x speed up when taking data transfer into account (data transfer time is ~2s while execution time is ~0.5s)
- Version 2 code is about 6x faster than version 1

02/05/2011

X. Lapillonne

15

Рис. 3.2: Сравнение производительности CPU и PGI Accelerator на примере блока облачной микрофизики (Xavier Lapillonne, Oliver Fuhrer)


## 3.3 PathScale HMPP / OpenHMPP

ФФ

## 3.4 f2c-acc

Компилятор f2c-acc (Mark Govett et al, NOAA) для маркированных директивами циклов заданного исходного кода на языке Fortran генерирует эквивалентный C/C++ код GPU-ядра и хост-драйвера (копирование данных, запуск ядра и тд). В целом метод выглядит как близкий аналог PGI Accelerator, но без привязки к компиляторам PGI (рис. 3.3-3.5). Даже если поддержка Fortran 95 реализована достаточно хорошо, в отношении f2c-acc справедлив тот же довод, что и для PGI Accelerator: избыточная ручная работа по определению зависимостей и параллельных циклов. В то же время доступность качественного открытого эквивалента повышает привлекательность решения PGI.

### F2C-ACC compiler



---


- F2C-ACC was developed at NOAA by the same team of SMS (Mark Govett et al.)...
- ....in order to reduce the porting-time of a legacy Fortran code to GPUs
- It works well with Fortran 77 codes plus some extension towards Fortran 95 (most of them!)
- F2C-ACC is an “open” project. Actual release is 3.0

<http://www.esrl.noaa.gov/gsd/ab/ac/F2C-ACC.html>

17

Рис. 3.3: Особенности компилятора f2c-acc (Piero Lanucara)

### F2C-ACC Workflow



---


- F2C-ACC translates Fortran code, with user added directives, in CUDA (relies on m4 library for interlanguages dependencies)
- **Some hand coding could be needed (see results)**
- Debugging and optimization Tips (e.g. Thread, block synchronization, out of memory, coalesce, occupancy....) are to be done manually

Compile and linking using CUDA libraries to create an executable to run

20

Рис. 3.4: Особенности компилятора f2c-acc (Piero Lanucara)

Plans for porting COSMO to GPUs at CASPUR



- A realistic strategy should take into account the Porting of the entire COSMO application in order to benefit of GPUs acceleration (Amdhal rule!)
- Tools like CUDA Fortran or PGI Accelerator are excellent competitors but vendor product (with positive and negative connotations)
- F2C-ACC is a good starting point but, at this moment, some hand-coding is required (e.g sum reduction adjustment for Himeno Benchmark): **HPC and CUDA expertise are needed**

28

Рис. 3.5: Выводы по компилятору f2c-acc (Piero Lanucara)

## 3.5 Специализированный язык шаблонов SCS

Supercomputing Systems AG предлагает ряд оптимизаций для CPU-версии модели COSMO (рис. 3.6):

- минимизация числа промежуточных массивов (вероятно, в конечном итоге это сводится к уменьшению общего числа хранимых массивов, что теоретически может ослабить зависимость производительности от скорости работы памяти)
- объединение циклов (вероятно, полезным в действительности может оказаться только объединение циклов, работающих со связанными данными, в противном случае можно ожидать, что положительный эффект будет нивелирован ухудшением кешируемости)
- наилучший порядок циклов

Динамическое ядро модели предполагается переписать на C++ с использованием специализированного API построения разностных схем, что позволит генерировать

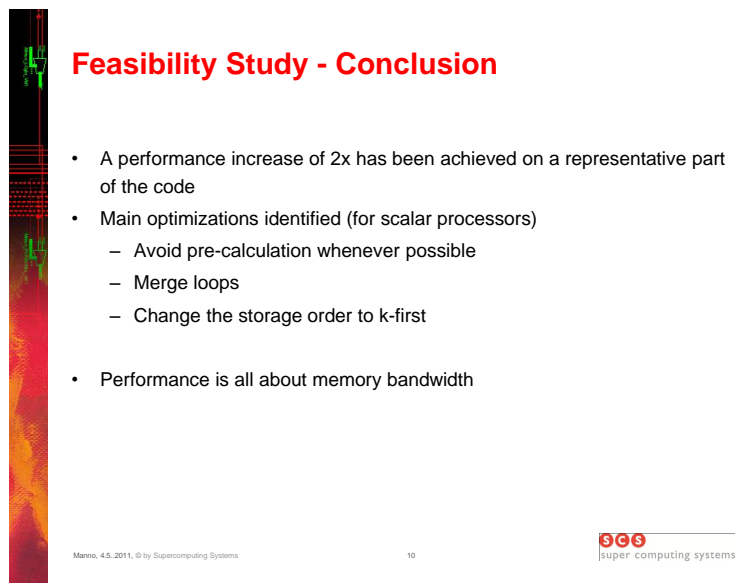



Рис. 3.6: Результаты исследования наиболее важных направлений оптимизации исходного кода модели COSMO (Tobias Gysi)

более эффективный машинный код и в перспективе добавлять различные реализации API, например GPU-бекенд (рис. 3.7-3.8). Очевидно эта задача выполнима: с помощью конструкций подобных STL может быть реализован какой-угодно API и весьма вероятно, что написанный с его помощью код может быть лучше оптимизирован компилятором. С другой стороны неясно, в какой мере такой подход сочетается с необходимостью постоянного развития модели и знаниями её основных разработчиков.

## 3.6 par4all

ФФ





## Stencil Library - Ideas


- It is challenging to develop a stencil library
  - There is no big chunk of work that can be hidden behind a API call (e.g. matrix multiplication)
  - The actual update function of the stencil is heavily application specific and performance critical
- We use a DSEL like approach (Domain Specific Embedded Language)
  - “Stencil language” embedded in C++
  - Separate description of **loop logic** and **update function**
  - During compile time generate optimized C++ code (possible due to C++ meta programming capabilities)

Manno, 4.5.2011, © by Superscomputing Systems

14

SCS  
super computing systems

Рис. 3.7: Идеи по разработке библиотеки, обобщающей программирование разностных схем (Tobias Gysi)



## Stencil Code – My Toy Example

### 1. Naive

```

for k
  a(k) := b(k) + c(k)
end
...

for k
  d(k) := a(k-1)*e(-1) +
          a(k)*e(0) +
          a(k+1)*e(+1)
end
...

for k
  f(k) := a(k)*g(k) + d(k)
end
          
```

### 5. Pre-calculation with stages & column Buffer

```

Stencil
  Stage 1
    a := b + c
  Stage 2
    d := a*e (k:-1,0,1)
  Stage 3
    f := a*g + d
Apply Stencil
          
```

Manno, 4.5.2011, © by Superscomputing Systems

22

SCS  
super computing systems

Рис. 3.8: Пример использования API построения разностных схем (справа), эквивалентного коду на языке Fortran (слева) (Tobias Gysi)

### 3.7 Обоснование выбора программной модели

В проектах РОМРА и других решениях можно выделить положительные и отрицательные стороны:

- Все решения требуют, чтобы параллельные циклы были определены пользователем вручную
- Все решения требуют, чтобы пересылки данных между CPU и GPU были определены пользователем вручную
- + Программные модели, основанные на добавлении директив обеспечивают обратную совместимость
- Несмотря на то, что компиляция кода для GPU происходит отдельно, коммерческие продукты также привязывают к собственным решениям и компиляцию кода на CPU
- Коммерческие решения трудно развивать и улучшать, имея доступ только к публичному интерфейсу
- Единственное открытое решение (f2c-асс) завязано на устаревший компилятор и может требовать значительных ресурсов для обеспечения поддержки современных стандартов языка Fortran

Таким образом, наиболее перспективным могло бы оказаться решение со следующими свойствами:

- Автоматическое выделение параллельных циклов
- Автоматическое определение зависимостей данных
- Использование оригинального исходного кода на языке Fortran без каких-либо изменений
- Возможность подключения любых компиляторов при сборки кода для CPU и GPU

- Максимально открытое решение, доступное в исходных кодах
- Использование актуальных разработок с хорошей поддержкой
- Производительность не хуже, чем у коммерческих решений
- Возможность проверки правильности результатов работы каждого отдельного GPU-ядра

По результатам анализа принято решение реализовать в рамках данной работы систему генерации GPU-кода, удовлетворяющую вышеперечисленным критериям.

## Глава 4

# Система генерации кода для GPU

ФФ

### 4.1 Трансформация исходного кода

Ключевой механизм в основе разрабатываемой системы — расщепление оригинального исходного кода на языке Fortran на части, предназначенные для исполнения на CPU и на GPU и их компиляция. Вообще говоря, не обязательно работать на уровне исходного кода, в некоторых других системах расщепление производится уже во внутреннем представлении компилятора AST (Abstract Syntax Tree). Но в таком случае решение либо жёстко привязывается к компилятору, либо восстанавливает программу из AST для дальнейшей передачи во внешний компилятор. В то же время трансляция кода в AST и генерация из него эквивалентного кода на другом языке является единственным способом решить задачу при отсутствии компилятора Fortran для GPU. Принимая во внимание требования по обеспечению поддержки внешних компиляторов и использованию открытых решений, возможны следующие варианты:

1. Использование AST открытого компилятора Fortran (например, `rose`, `gfortran`, `g95` или `pathscale`) для преобразования исходного кода, поддержка компиляции для CPU и GPU внешними компиляторами

2. Использование AST открытого компилятора Fortran Open64 для преобразования исходного кода, поддержка компиляции для CPU внешним компилятором, компиляция для GPU напрямую из AST с помощью open64-совместимого компилятора nvorencs
3. Использование LLVM, преобразование кода на уровне IR, восстановление эквивалентного кода на C для GPU, компиляция с помощью LLVM или восстановление кода для внешнего компилятора — для CPU
4. Генерация разметки исходного кода (pretty-printed AST) с помощью открытого компилятора и преобразование его независимыми средствами, например, преобразование XML-разметки исходного кода
5. Использование f2c или f2c-асс для преобразования кода на языке Fortran в C-код.

Первый вариант выглядит наиболее привлекательно в сочетании с gfortran, g95 или pathscale, так как основан на использовании широко поддерживаемых компиляторов (обращаться с g95 сложнее, кроме того в нём отсутствует поддержка ряда необходимых возможностей современного стандарта языка Fortran, например, inline-функций). Вместе с тем, развивать его слишком трудно без участия специалистов соответствующего профиля. Возможно, по мере готовности системы и в случае наличия ресурсов, имеет смысл переключиться на этот вариант.

Второй вариант является наиболее дружественным по отношению к средствам, входящим в состав NVIDIA CUDA Toolkit. Теоретически возможно добиться бесшовной компиляции CUDA-кода для GPU в дополнение к преимуществам первого варианта. К сожалению, развивать его бесперспективно, так как nvorencs уже сейчас слабо поддерживается, а в новых выпусках CUDA SDK будет постепенно заменён на nvvm.

Третий вариант делает ставку на LLVM — проект с коммерчески-совместимой свободной лицензией, имеющий в последние годы исключительный успех. Основная проблема при его использовании — внутреннее представление наподобие ассемблера

(LLVM IR), из-за которого в восстановленном C-коде теряется много высокоуровневой информации, например оригинальный вид циклов. Качество C-backend тем не менее постоянно улучшается. Привлекательная возможность LLVM — генерация эквивалентного C-кода для любого языка, поддерживаемого GNU-компилятором (C++, Fortran, Ada, Go), что теоретически позволяет перевести на CUDA множество программ.

Четвёртый вариант позволяет получить разбор исходного кода с помощью компонентов компилятора, но при этом работать с простым текстовым представлением AST. К началу проекта по расщеплению кода данным методом уже имелись значительные наработки, используя которые можно было быстрее всего получить необходимый результат.

Пятый вариант — f2c — фактически, надёжно поддерживает только Fortran 77, что не подходит для целевых моделей. Доработка f2c-асс, вероятно, возможна, но требует глубокого анализа.

На первом этапе проекта было принято решение реализовать метод трансформации исходного кода, сочетающий третий и четвёртый вариант: расщепление кода на части для CPU и GPU происходит с помощью разбора исходного кода с XML-разметкой, генерацию CUDA-кода для GPU производит модифицированный LLVM C-backend.

## 4.2 CUDA-backend на основе LLVM

ФФ

## 4.3 Runtime-библиотека

ФФ

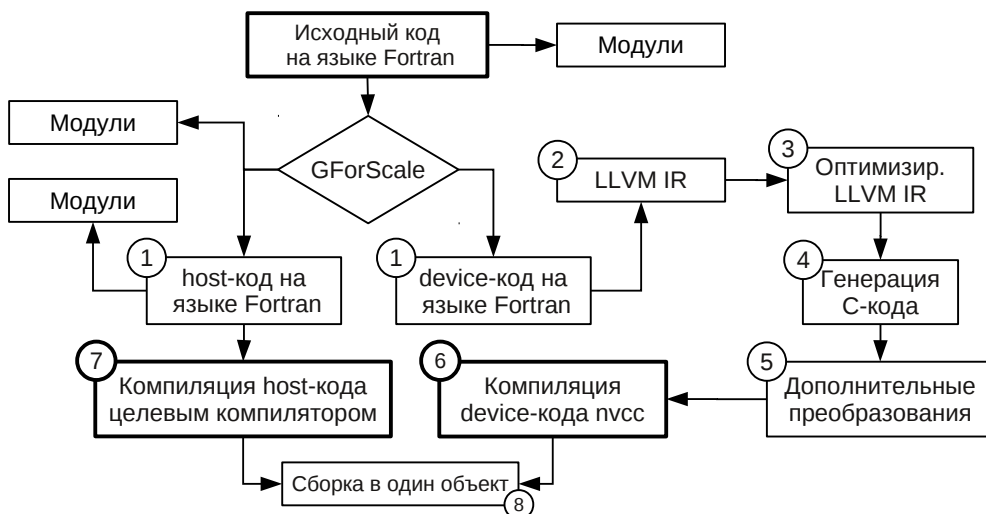


Рис. 4.1: Общая схема генератора

## 4.4 Общая схема генератора

Общая схема этапов генерации GPU-кода представлена на рис. 4.1. Рассмотрим их работу на примере теста axpy.

```

1. /opt/kgen/bin/kgen-gforscale
   -Wk,--gforscale-scene-path=/opt/kgen/transforms/split/
   -Wk,--gforscale-specific-mode=default
   -Wk,--gforscale-markup-mode=tree -g -c axpy.f90 -D__CUDA_DEVICE_FUNC__

```

Из обычного исходного кода на языке Fortran генерируются части эквивалентного кода на языках Fortran и C++ (части на C++ являются вспомогательными). Одни части кода подлежат исполнению на CPU, другие - на GPU. Разделение кода на части производит скрипт kgen-gforscale, используя преобразования внутреннего XML-представления исходного кода.

```

/opt/llvm/dragonegg/bin/dragonegg-gfortran -g -c axpy.host.F90
-D__CUDA_DEVICE_FUNC__ -ffree-line-length-none -I/opt/kgen/include -o
axpy.host.F90.o

```

Перед следующим шагом часть кода для CPU проходит через компилятор `/opt/llvm/dragonegg/bin/dragonegg-gfortran` для получения mod-файлов соответствующей версии. Они необходимы, так как содержат используемую Fortran-фронтом информацию о данных, совместно используемых несколькими объектами. Формат mod-файлов слабо стандартизирован, поэтому они должны точно соответствовать компилятору.

2. 

```
/opt/llvm/dragonegg/bin/dragonegg-gfortran -c
axy.axy_loop_1_gforscale.device.F90 -D__CUDA_DEVICE_FUNC__
-ffree-line-length-none
-fplugin=/opt/llvm/dragonegg/lib64/dragonegg.so -O0 -S
-fplugin-arg-dragonegg-emit-ir -o axy.axy_loop_1_gforscale.device.F90.bc
```

Части исходного кода на языке Fortran, предназначенные для GPU преобразуются в LLVM IR (Low Level Virtual Machine Intermediate Representation) с помощью компилятора `gfortran` и плагина `dragonegg`.

3. 

```
/opt/llvm/bin/opt -std-compile-opts
axy.axy_loop_1_gforscale.device.F90.bc -S -o
axy.axy_loop_1_gforscale.device.F90.bc.opt
```

Оптимизирующий проход LLVM, благодаря которому генерируемый на следующем шаге CUDA-код становится более эффективным и удобным для чтения. Процессы анализа и оптимизации IR-кода в LLVM состоят из проходов (passes), анализирующих или трансформирующих код (т.е. получающих новую версию IR).

4. 

```
/opt/llvm/bin/llc -march=c axy.axy_loop_1_gforscale.device.F90.bc.opt
-o axy.axy_loop_1_gforscale.device.F90.bc.cu
```

Генерация CUDA-кода из LLVM IR с помощью модифицированного LLVM C-Backend (файл `llvm/lib/Target/CBackend/CBackend.cpp`).



5. Поиск и замена фрагментов исходного кода с подстановкой атрибута `__global__` в определение основной функции-ядра и реализаций `__device__`-функций, отображающих индексы циклов на индексы блоков GPU. Этот шаг не имеет специальной команды, его выполняет скрипт `kgen` с помощью регулярных выражений.

```
$code =~ s/\#ifdef\s__CUDA_DEVICE_FUNC__\n__device__\n#endif\n
void\s$name\_\\(\#ifdef__CUDA_DEVICE_FUNC__\nextern "C"
__global__\n#endif\n#define $name\_ $name\nvoid $name\_(/s;

$code =~ s/void\s$name\_blockidx_x\(unsigned\sint\s*,\sunsigned\sint\s,
\sunsignedint\s\\)\;/void $name\_blockidx_x(unsigned int* index,
unsigned int start, unsigned int end)
{ *index = blockIdx.x + start; }/s;

$code =~ s/void\s$name\_blockidx_y\(unsigned\sint\s*,\sunsigned\sint\s,
\sunsigned int\s\\)\;/void $name\_blockidx_y(unsigned int* index,
unsigned int start, unsigned int end)
{ *index = blockIdx.y + start; }/s;

$code =~ s/void\s$name\_blockidx_z\(unsigned\sint\s*,\sunsigned\sint\s,
\sunsigned int\s\\)\;/void $name\_blockidx_z(unsigned int* index,
unsigned int start, unsigned int end)
{ *index = threadIdx.z + start; }/s;
```

6. `nvcc -g -c axpy.axpy_loop_1_gforscale.device.F90.bc.cu`  
`-D__CUDA_DEVICE_FUNC__ --ptxas-options="-v" -arch=sm_20 -o`  
`axpy.axpy_loop_1_gforscale.device.F90.o`

Сборка части кода для GPU с помощью компилятора CUDA.

7. `gfortran -g -c axpy.host.F90 -D__CUDA_DEVICE_FUNC__`  
`-ffree-line-length-none -I/opt/kgen/include -o axpy.host.F90.o`

```
gfortran -g -c axpy.axpy_loop_1_gforscale.host.F90
-D__CUDA_DEVICE_FUNC__ -ffree-line-length-none -I/opt/kgem/include -o
axpy.axpy_loop_1_gforscale.host.F90.o

g++ -g -I/opt/kgem/include -c axpy.axpy_loop_1_gforscale.host.cpp -o
axpy.axpy_loop_1_gforscale.host.cpp.o
```

Сборка частей кода для CPU - основного и вспомогательных объектов.

```
8. ld --unresolved-symbols=ignore-all -r -o axpy.o_kgem axpy.host.F90.o
axpy.axpy_loop_1_gforscale.host.F90.o
axpy.axpy_loop_1_gforscale.host.cpp.o
axpy.axpy_loop_1_gforscale.device.F90.o
```

Сборка объектов, соответствующих различным частям кода в единый объектный файл. За счёт этого результат работы генератора с точки зрения компоновки программных единиц идентичен обычному компилятору.

```
9. objcopy --wildcard
--localize-symbol=__device_stub__*
--localize-symbol=_gforscale_blockidx_*
--localize-symbol=_gforscale_threadidx_*
--localize-symbol=_gforscale_desc_*
--localize-symbol=_gforscale_init*
--localize-symbol=_gforscale_free*
--localize-symbol=_loop*_gforscale
--localize-symbol=_gforscale_compare_ axpy.o_kgem axpy.o
```

Окончательная генерация результирующего объектного файла с переводом автоматически созданных функций в локальную область видимости для устранения потенциальных конфликтов имён на этапе линковки. Вместе с этим происходит перезапись объекта ранее созданного обычным компилятором, который

был бы использован в случае, если генерация GPU-ядер по какой-либо причине не отработала корректно.

## Глава 5

# Генерация GPU-кода для модели COSMO

ФФ

## Глава 6

# Руководство пользователя

Генератор GPU-ядер называется `kgen-gfortran` и в целом ведёт себя как обычный компилятор программ на языке Fortran, работать с ним столь же просто. Необходимые программы установлены на GPU-сервере СибНИГМИ и готовы к использованию. В данном разделе содержится описание опций компилятора и настроек среды, приведены примеры использования системы для компиляции моделей COSMO и WRF.

### 6.1 Опции компилятора

Компилятор `kgen-gfortran` вводит ряд дополнительных опций с префиксом `-Wk` для управления генерацией кода на GPU:

Опция	Значение	Комментарий
-Wk,--bypass	—	Отключение генерации GPU-ядер (компиляция только для CPU)
-Wk,--host-compiler=	gfortran	Имя внешнего компилятора для кода на CPU
-Wk,--kernel-compiler=	nvcc	Имя внешнего компилятора для GPU-ядер
-Wk,--kernel-target=	cuda	Программная модель для GPU
-Wk,--specific-mode=	cosmo	Специальный режим для компиляции определённой программы (эта опция будет удалена после добавления поддержки Polly)

## 6.2 Настройка среды

Компилятор kgen-gfortran обычно устанавливается в папку */opt/kgen/bin*, программы собранные с его помощью зависят от динамической библиотеки из */opt/kgen/lib/*. Подключить эти пути можно соответственно через переменные окружения `PATH` и `LD_LIBRARY_PATH`:

```
$ export PATH=$PATH:/opt/kgen/bin
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/kgen/lib
```

Поведение программы, собранной kgen-gfortran можно настроить с помощью следующих переменных среды:

Переменная	Значение	Комментарий
gforscale_runmode	1, 2, 3	Режим исполнения для всех циклов: 1 - на CPU (по умолчанию), 2 - на GPU с поддержкой CUDA, 3 - и на CPU, и на CUDA GPU + сравнение результатов
gforscale_debug_output	0, 1, 2, 3	Отладочный вывод из функций runtime-библиотеки (по умолчанию 0 - отключён)
gforscale_error_output	0, 1, 2, 3	Вывод ошибок из функций runtime-библиотеки (по умолчанию 0 - всё включено)
%s_loop_%d_gforscale	1, 2, 3	Режим исполнения конкретного цикла с номером %d в функции %s: 1 - на CPU (по умолчанию), 2 - на GPU с поддержкой CUDA, 3 - и на CPU, и на CUDA GPU + сравнение результатов

## 6.3 Компиляция COSMO

```
svn co file:///svn/cosmo
```

Переместившись в папку с исходным кодом компонентов модели, необходимо запустить make-скрипт с указанием флагов, подключающих генерацию GPU-ядер:

```
cd cosmo/source
make HAVE_KERNELGEN=1 HAVE_KERNELGEN_CUDA=1
```

Обычно компиляция занимает продолжительное время. Если на системе доступно  $N > 1$  ядер, то сборку можно ускорить за счёт параллельной компиляции добавлением опции `-jN`.

## 6.4 Компиляция WRF

Репозиторий с исходным кодом модели WRF версии 3.3 находится на GPU-сервере СибНИГМИ. По сравнению с официальным релизом, в нём сделано одно исправление, также kgen-gfortran добавлен в список поддерживаемых компиляторов. Любой пользователь системы может сделать локальную копию репозитория:

```
svn co file:///svn/wrf
```

Переместившись в корневую папку файлов модели, необходимо запустить скрипт конфигурации с указанием пути к библиотекам NetCDF:

```
cd wrf/  
NETCDF=/opt/openmpi_kgen-1.4.4/ ./configure
```

Диалог выбора конфигурации будет содержать варианты сборки с использованием kgen-gfortran:

```
19.  Linux x86_64, kgen-gfortran generator for CUDA with gcc   (serial)  
20.  Linux x86_64, kgen-gfortran generator for CUDA with gcc   (smpar)  
21.  Linux x86_64, kgen-gfortran generator for CUDA with gcc   (dmpar)  
22.  Linux x86_64, kgen-gfortran generator for CUDA with gcc   (dm+sm)
```

Далее сборка производится скриптом `./compile` с указанием одной из предопределённых конфигураций модели:

```
./compile em_tropical_cyclone
```

Обычно компиляция занимает продолжительное время. Если на системе доступно  $N > 1$  ядер, то сборку можно ускорить за счёт параллельной компиляции добавлением опции `-jN`.



## 6.5 Ограничения

Компилятор `kgen-gfortran` может быть недостаточно хорошо совместим с некоторыми сложными `configure`-скриптами. При необходимости выполнения `configure`-скрипта следует переключить значение переменной `bypass` в файле `/opt/kgen/bin/kgen` на 1, а после завершения конфигурации — вернуть обратно 0.

## Глава 7

# Дополнительные сведения для разработчиков

### 7.1 Компиляция компонентов системы

ФФ

#### 7.1.1 LLVM

Загрузка исходного кода (последняя протестированная ревизия — 134365):

```
svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm -r134365
cd llvm/
```

Наложение двух патчей, исправляющих две открытые проблемы LLVM C-backend и третьего — для обеспечения генерации CUDA-кода.

```
patch -d . -p1 < ../kernelgen/patches/llvm/varargs.patch
patch -d . -p1 < ../kernelgen/patches/llvm/inlasm.patch
patch -d . -p1 < ../kernelgen/patches/llvm/nvcc.patch
```

Сборка и установка LLVM:

```
mkdir build
cp -rf include/ build/include/
cd build
../configure --enable-jit --enable-optimized --enable-shared --prefix=/opt/llvm
--enable-targets=host,cbe
make -j12 CXXFLAGS=-O3
sudo make install
```

В опциях указан путь для установки */opt/llvm* и поддержка архитектур host (архитектура данной машины) и cbe (C backend).

Добавить *llvm* в пути можно стандартным образом — через *PATH* и *LD\_LIBRARY\_PATH* или */etc/profile.d/* и */etc/ld.so.conf.d/*:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/llvm/lib:/opt/llvm/lib64/
export PATH=$PATH:/opt/llvm/bin
```

```
$cat /etc/ld.so.conf.d/llvm.conf
/opt/llvm/lib
$cat /etc/profile.d/llvm.sh
export PATH=$PATH:/opt/llvm/bin
```

### 7.1.2 GNU GCC 4.5

Зависимости для системы Fedora:

- gmp-devel, mpfr-devel, libmpc-devel
- elfutils-libelf-devel
- flex
- glibc-devel.i686

Зависимости для системы Debian / Ubuntu:

- libgmp3-dev, libmpfr-dev, libmpc-dev
- libelf-dev
- flex
- libc6-dev, libc6-dev-i386
- gcc-multilib

Загрузка исходного кода (последняя протестированная ревизия — 174313), сборка и установка:

```
svn co http://llvm.org/svn/llvm-project/dragonegg/trunk dragonegg
svn co http://gcc.gnu.org/svn/gcc/branches/gcc-4_5-branch gcc-4.5 -r174313
cd gcc-4.5/
patch -d . -p1 < ../kernelgen/patches/gcc/gcc.patch
mkdir build
cd build/
../configure --prefix=/opt/llvm/dragonegg/ --program-prefix=dragonegg-
--enable-languages=c,c++,fortran --with-mpfr-include=/usr/include/
--with-mpfr-lib=/usr/lib64 --with-gmp-include=/usr/include/
--with-gmp-lib=/usr/lib64 --enable-plugin
make -j12
sudo make install
```

## 7.2 DragonEgg

## Глава 8

### Лицензионный статус

Реализация, полученная в данном проекте является надстройкой над множеством различных программ. В данном разделе собраны сведения о лицензионном статусе используемых компонентов. Все они так или иначе доступны в исходных кодах, за исключением части `cuda toolkit`.

Название	Лицензия	Комментарий
<code>llvm</code>	BSD	Инфраструктура компилятора LLVM
<code>gfortran</code>	GPL	Компилятор GNU Fortran
<code>dragonegg</code>	BSD	Плагин, связывающий <code>llvm</code> и <code>gfortran</code>
<code>g95</code>	GPL	Компилятор Fortran
<code>g95xml</code>	GPL	XML-разметка исходного кода на основе компилятора <code>g95</code>
<code>libxml/libxslt</code>	MIT	Библиотека для работы с XML и XSLT
<code>nvopencc</code>	GPL	Вариант компилятора <code>open64</code> с поддержкой генерации GPU-ядер на CUDA
<code>cuda toolkit</code>	NVIDIA	Инфраструктура компилятора NVIDIA CUDA

# Глава 9

## План развития

В этом разделе дано описание конкретных технических подзадач, соответствующих общему плану работ.

### 9.1 Первый этап (окончен)

1. Реализация программы-драйвера по типу gсс или nvсс, организующей запуск приложений, ответственных за отдельные этапы компиляции
2. Тестирование генерации кода на CUDA C с помощью LLVM C Backend, обсуждение существенных проблем с разработчиками LLVM, написание патчей
3. Доработка созданной в 2009 г. системы расщепления исходного кода на части для CPU и GPU на основе g95-xml
4. Разработка runtime-библиотеки:
  - (a) Запуск ядер и упаковка аргументов с помощью CUDA API
  - (b) Синхронизация данных двумя способами: явно (cudaMalloc/cudaMemcpy) и memory mapping (zero-copy)
  - (c) Поддержка синхронизации allocatable-переменных и данных в модулях

5. Разработка системы контроля правильности результатов на уровне отдельных ядер:
  - (а) Запуск ядра с предварительной дубликацией входных и выходных данных
  - (б) Генерация функции сравнения результатов для каждого ядра
6. Сборка модели COSMO с помощью генератора, представление результатов корректной работы большого множества GPU-ядер
7. Реализация набора базовых тестов для автоматизированного контроля регрессивных изменений

## 9.2 Второй этап

1. Улучшение поддержки/совместимости генератора с различными моделями
2. Улучшение производительности/эффективности
3. Сравнение производительности с другими системами и результатами проектов РОМРА

### 9.2.1 Задачи поддержки/совместимости

1. Тестирование сборки COSMO и других моделей конечными пользователями, отчёты об ошибках (участники от СибНИГМИ)
  - (а) Проверка сборки рабочей версии модели WRF с помощью kgen-gfortran
  - (б) Проверка сборки модели ПЛАВ, предварительное переключение на gcc, если текущая версия настроена на компилятор Intel (совместимость с Intel не тестировалась)
2. Исправление наиболее существенных ошибок и недоработок в системе разделения кода
  - (а) Решение проблем с использованием производных типов (структур)

3. Использование Polly/Pluto для точного определения параллельных циклов
  - (a) Освоение pluto и polly для LLVM на простых тестах
  - (b) Проверка возможности добавления шага polly LLVM перед генерацией CUDA-кода
  - (c) Освоение методов реконструкции данных о циклах из представления LLVM IR, проверка возможности получения этой информации после прохода polly и её использования в конфигурации грида GPU.
4. Частичная докомпиляция ядер с внешними зависимостями на этапе линковки
  - (a) Выбор метода поиска внешних зависимостей
  - (b) Поддержка хранения метаданных в специальной секции объектных файлов
  - (c) Реализация скрипта линковки kgen-ld с комбинированием исходного кода нескольких зависимых объектов в единый объект и перекомпиляцией

### 9.2.2 Задачи производительности/эффективности

1. Использование разделяемой памяти в шаблонах численных схем (Александр Мыльцев)
  - (a) Изучение литературы по существующим генераторам на основе ROSE или Polly
  - (b) Добавление простой реализации в преобразования XSLT
2. Уменьшение времени запуска GPU-ядер (кеширование конфигураций)
  - (a) Перенос определения статических зависимостей GPU-ядра в секцию инициализации
  - (b) Решение проблем с использованием `__attribute__((constructor))` в CUDA или проработка обходного пути



### 3. Уменьшение времени обмена данными

- (a) (вариант) Портинг на GPU также и последовательного кода в режиме одного потока (теоретически, значительно меньше обменов, остаётся только вывод и MPI)
  - i. Имеет смысл только при отличном качестве работы системы разделения кода
  - ii. Установка device-заглушек на вызовы функций gfortran runtime library
  - iii. Генерация GPU-ядер для последовательных циклов в режиме одного потока
  - iv. Реализация некоторых функций gfortran runtime library для GPU (работа с памятью, печать, вывод)
- (b) (вариант) Поддержка выборочной/отложенной синхронизации данных на основе статического графа зависимостей (гибкий вариант)
  - i. Подключение g95xml-refids в систему разделения кода, реализация правила статического поиска символов, зависящих от символов в ядре, установка точек обязательной синхронизации данных
- 4. (предложение С. В. Ковылова) Реализация гибридного разделения вычислений между CPU и GPU

## 9.3 Предложения координационного характера

1. Организация официального хостинга проекта в СибНИГМИ, переезд проекта с `tesla.parallel.ru` (возможно, на `sscc`)
2. Согласование возможных вариантов тестирования multi-GPU режима (подача заявки на доступ к одному из GPU-кластеров МГУ, разрешение на тестирование в NVIDIA и др.)
3. Разрешение на контакт с участниками COSMO/POMPA с целью представления результатов на ближайшей встрече разработчиков (сентябрь 2011, Рим)