Технологии автоматического распараллеливания кода для GPU

Дмитрий Микушин Николай Лихогруд Eddy Z. Zhang
Christopher Bergström Сергей Ковылов

Семинар «Суперкомпьютерные технологии в науке, образовании и промышленности»

Простейший пример: Fortran

```fortran
program demo

integer :: argc, nx, ny, ns
character(len=32) :: arg
real, allocatable, dimension(:,:,:) :: x, y, xy
real :: start, finish

! Read arguments
call get_command_argument(1, arg)
read(arg, '(I32)') nx
call get_command_argument(2, arg)
read(arg, '(I32)') ny
call get_command_argument(3, arg)
read(arg, '(I32)') ns

! Allocate data arrays.
allocate(x(nx,ny,ns), y(nx,ny,ns), xy(nx,ny,ns))

! Initialize arrays.
x = atan(1.0)
y = x

! Computational loop
call cpu_time(start)
do k = 1, ns
  do j = 1, ny
    do i = 1, nx
      xy(i,j,k) = asin(sin(x(i,j,k))) + acos(cos(y(i,j,k)))
    enddo
  enddo
enddo
call cpu_time(finish)

write(*,*) 'compute time = ', finish - start

write(*,*) 'maxval = ', maxval(xy), &
  'minval = ', minval(xy)

! Deallocate arrays.
deallocate(x, y, xy)

end program demo
```

Простейший пример: Fortran

Пример компилируется как обычно, но с использованием *kernelgen-gfortran* вместо *gfortran*:

```
$ kernelgen-gfortran -O3 example_f.f90 -o example_f
```

KernelGen всегда генерирует гибридный бинарный файл, который может работать как на CPU, так и на GPU. CPU-версия запускается обычным образом:

```
$ ./example_f 512 256 256
 compute time =    1.8481150
 maxval =    1.5707963     minval =    1.5707963
```

Для запуска GPU-версии, достаточно установить значение переменной окружения *kernelgen_runmode* равным 1:

```
$ kernelgen_runmode=1 ./example_f 512 256 256
 compute time =    0.28801799
 maxval =    1.5707964     minval =    1.5707964
```

Простейший пример: C

```c
#include <malloc.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int main(int argc, char* argv[]) {

int nx = atoi(argv[1]);
int ny = atoi(argv[2]);
int ns = atoi(argv[3]);

size_t szarray = nx * ny * ns;
size_t szarrayb = szarray * sizeof(float);

float* x = (float*)malloc(szarrayb);
float* y = (float*)malloc(szarrayb);
float* xy = (float*)malloc(szarrayb);

for (int i = 0; i < szarray; i++)
{
  x[i] = atan(1.0);
  y[i] = x[i];
}
struct timeval start, finish;
```

```c
gettimeofday(&start, NULL);
for (int k = 0; k < ns; k++)
  for (int j = 0; j < ny; j++)
    for (int i = 0; i < nx; i++)
    {
      int idx = i + nx * (j + ny * k);
      xy[idx] = asinf(sinf(x[idx])) + acosf(cosf(y[idx]));
    }
gettimeofday(&finish, NULL);
printf("compute time = %f\n",
  get_time_diff(&start, &finish));

float minval = xy[0], maxval = xy[0];
for (int i = 0; i < szarray; i++)
{
  if (minval > xy[i]) minval = xy[i];
  if (maxval < xy[i]) maxval = xy[i];
}
printf("maxval = %f, minval = %f\n", maxval, minval);

// Deallocate arrays.
free(x); free(y); free(xy);

return 0;
}
```

Простейший пример: C

Пример компилируется как обычно, но с использованием *kernelgen-gcc* вместо *gcc*:

```
$ kernelgen-gcc -O3 -std=c99 example_c.c -o example_c
```

KernelGen всегда генерирует гибридный бинарный файл, который может работать как на CPU, так и на GPU. CPU-версия запускается обычным образом:

```
$ ./example_c 512 256 256
compute time = 1.848575
maxval = 1.570796, minval = 1.570796
```

Для запуска GPU-версии, достаточно установить значение переменной окружения *kernelgen_runmode* равным 1:

```
$ kernelgen_runmode=1 ./example_c 512 256 256
compute time = 0.293359
maxval = 1.570796, minval = 1.570796
```

Цели и предпосылки проекта KernelGen

Цели:

- Портировать приложения на GPU без изменения оригинального исходного кода, генерируя GPU-код на внутреннем уровне
- Средство портирования должно быть совместимо с наиболее важными существующими вычислительными приложениями (не ставится цель делать универсальное распараллеливание!)

Предпосылки:

- Традиционные языки программирования вполне могут быть сохранены, если дополнить компилятор средствами более глубокого анализа кода
- Применение OpenACC в сложных приложениях затруднено множеством ограничений
- В следующих поколениях вычислительных систем GPU вполне может занять более центральную роль, что несовместимо с offload-моделью OpenACC

Потенциал применения

Численные модели:

- Климат/погода, CFD
- Сейсмология
- ...

Потенциал применения

Численные модели:

- Климат/погода, CFD
- Сейсмология
- ...

Коммерческие разработки:

- **Bank of America Merrill Lynch**: Суперкомпиляция GPU-ядер для Монте-Карло с множеством сценариев, на основе LLVM
- **D.E. Shaw Research**: оптимизация PTX-кода, программные вычисления с фиксированной точкой на GPU
- **Qualcomm**: Автораспараллеливание, автовекторизация, на основе LLVM/Polly

Основные ограничения OpenACC

Актуальная версия стандарта OpenACC в основном пригодна для ускорения малых и небольших приложений. Более широкое использование OpenACC затруднено следующими ограничениями:

- **Внешние вызовы** – поддержка внешних вызовов в циклах, портируемых на ускоритель, необходима в больших проектах, в которых функциональность распределена между множеством объектов, например, модулями в Fortran.
- **Анализ зависимостей указателей** – во многих ситуациях компилятор не может однозначно определить наличие или отсутствие взаимосвязи между заданными указателями, требуя дополнительных указаний со стороны пользователя.
- **Регионы данных** – необходимость указывать какие данные должны постоянно находиться в памяти GPU

$\Rightarrow$ Насколько оправдано жертвовать дизайном и интегрированностью исходного кода ради его ускорения?

OpenACC: внешние вызовы невозможны

OpenACC не поддерживает вызов функций, определенных в других объектах:

sincos.f90

```
!$acc parallel
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy(i, j, k) = sincos_ijk(x(i, j, k), y(i, j, k))
    enddo
  enddo
enddo
!$acc end parallel
```

function.f90

```
function sincos_ijk(x, y)

  implicit none
  real, intent(in) :: x, y
  real :: sincos_ijk

  sincos_ijk = sin(x) + cos(y)

end function sincos_ijk
```

```
pgfortran -fast -Mnomain -Minfo=accel -ta=nvidia,time -Mcuda=keepgpu,keepbin,keepptx,↩
     ptxinfo -c ../sincos.f90 -o sincos.o
PGF90-W-0155-Accelerator region ignored; see -Minfo messages  (../sincos.f90: 33)
sincos:
     33, Accelerator region ignored
     36, Accelerator restriction: function/procedure calls are not supported
     37, Accelerator restriction: unsupported call to sincos_ijk
```

KernelGen: внешние вызовы возможны

Разрешение зависимостей во время линковки
Генерация ядер во время исполения

$\Rightarrow$

Поддержка внешних вызовов из других объектов или статических библиотек

sincos.f90

```
!$acc parallel
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy(i, j, k) = sincos_ijk(x(i, j, k), y(i, j, k))
    enddo
  enddo
enddo
!$acc end parallel
```

function.f90

```
function sincos_ijk(x, y)

  implicit none
  real, intent(in) :: x, y
  real :: sincos_ijk

  sincos_ijk = sin(x) + cos(y)

end function sincos_ijk
```

```
Launching kernel __kernelgen_sincos__loop_3
    blockDim = { 32, 4, 4 }
    gridDim = { 16, 128, 16 }
Finishing kernel __kernelgen_sincos__loop_3
__kernelgen_sincos__loop_3 time = 4.986428e-03 sec
```

KernelGen: ... и из статических библиотек

Благодаря измененному LTO-обработчику, KernelGen может обрабатывать ядра с зависимостями из статических библиотек:

```
kernelgen-gcc -std=c99 -c ../main.c -o main.o
kernelgen-gfortran -c ../sincos.f90 -o sincos.o
kernelgen-gfortran -c ../function.f90 -o function.o
ar rcs libfunction.a function.o
kernelgen-gfortran main.o sincos.o -o function -L. -lfunction
```

```
$ kernelgen_runmode=1 ./function 512 512 64
__kernelgen_sincos__loop_3: regcount = 22, size = 512
Loaded '__kernelgen_sincos__loop_3' at: 0xc178f0
Launching kernel __kernelgen_sincos__loop_3
    blockDim = { 32, 4, 4 }
    gridDim = { 16, 128, 16 }
Finishing kernel __kernelgen_sincos__loop_3
__kernelgen_sincos__loop_3 time = 4.974710e-03 sec
```

OpenACC: слабый анализ указателей

Компилятор не может однозначно определить взаимосвязь между заданными указателями:

sincos.c

```
void sincos(int nx, int ny, int nz, float* x, float* y, float* xy) {
    #pragma acc parallel
    for (int k = 0; k < nz; k++)
        for (int j = 0; j < ny; j++)
            for (int i = 0; i < nx; i++) {
                int idx = i + nx * j + nx * ny * k;
                xy[idx] = sin(x[idx]) + cos(y[idx]);
            }
}
```

```
pgcc -fast -Minfo=accel -ta=nvidia,time -Mcuda=keepgpu,keepbin,keepptx,ptxinfo -c ../sincos.c -o sincos.o
PGC-W-0155-Compiler failed to translate accelerator region (see -Minfo messages): Could not find allocated-variable index ⟷
      for symbol (../sincos.c: 27)
sincos:
     27, Accelerator kernel generated
     28, Complex loop carried dependence of *(y) prevents parallelization
         Complex loop carried dependence of *(x) prevents parallelization
         Complex loop carried dependence of *(xy) prevents parallelization
...
     30, Accelerator restriction: size of the GPU copy of xy is unknown
```

KernelGen: точный анализ указателей

В KernelGen анализ указателей проводится во время исполнения, после подстановки значений:

```
for (c2=0;c2<=63;c2++) {
  for (c4=0;c4<=511;c4++) {
    for (c6=0;c6<=511;c6++) {
      Stmt__5_cloned_(c2,c4,c6);
    }
  }
}
```

```
Statements {
     Stmt__5_cloned_
            Domain      := { Stmt__5_cloned_[i0, i1, i2] : i0 >= 0 and i0 <= 63 and i1 >= 0 and i1 <= 511 and i2 >= 0 and ↩
                  i2 <= 511 };
            Scattering  := { Stmt__5_cloned_[i0, i1, i2] -> scattering[0, i0, 0, i1, 0, i2, 0] };
            ReadAccess  := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47246749696 + 1048576i0 + 2048i1 + 4i2 and o0↩
                  <= 47246749699 + 1048576i0 + 2048i1 + 4i2 };
            ReadAccess  := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47313862656 + 1048576i0 + 2048i1 + 4i2 and o0↩
                  <= 47313862659 + 1048576i0 + 2048i1 + 4i2 };
            WriteAccess := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47380975616 + 1048576i0 + 2048i1 + 4i2 and o0↩
                  <= 47380975619 + 1048576i0 + 2048i1 + 4i2 };
}
```

KernelGen: точный анализ указателей

В KernelGen анализ указателей проводится во время исполнения, после подстановки значений:

```
for (c2=0;c2<=63;c2++) {
  for (c4=0;c4<=511;c4++) {
    for (c6=0;c6<=511;c6++) {
      Stmt__5_cloned_(c2,c4,c6);
    }
  }
}
```

Статический поток управления (SCoP) для заданного вычислительного цикла

```
Statements {
     Stmt__5_cloned_
            Domain      := { Stmt__5_cloned_[i0, i1, i2] : i0 >= 0 and i0 <= 63 and i1 >= 0 and i1 <= 511 and i2 >= 0 and ↩
                  i2 <= 511 };
            Scattering  := { Stmt__5_cloned_[i0, i1, i2] -> scattering[0, i0, 0, i1, 0, i2, 0] };
            ReadAccess  := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47246749696 + 1048576i0 + 2048i1 + 4i2 and o0↩
                  <= 47246749699 + 1048576i0 + 2048i1 + 4i2 };
            ReadAccess  := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47313862656 + 1048576i0 + 2048i1 + 4i2 and o0↩
                  <= 47313862659 + 1048576i0 + 2048i1 + 4i2 };
            WriteAccess := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47380975616 + 1048576i0 + 2048i1 + 4i2 and o0↩
                  <= 47380975619 + 1048576i0 + 2048i1 + 4i2 };
}
```

KernelGen: точный анализ указателей

В KernelGen анализ указателей проводится во время исполнения, после подстановки значений:

```
for (c2=0;c2<=63;c2++) {
  for (c4=0;c4<=511;c4++) {
    for (c6=0;c6<=511;c6++) {
      Stmt__5_cloned_(c2,c4,c6);
    }
  }
}
```

Статический поток управления (SCoP) для заданного вычислительного цикла

```
Statements {
     Stmt__5_cloned_
            Domain      := { Stmt__5_cloned_[i0, i1, i2] : i0 >= 0 and i0 <= 63 and i1 >= 0 and i1 <= 511 and i2 >= 0 and ↩
                  i2 <= 511 };
            Scattering  := { Stmt__5_cloned_[i0, i1, i2] -> scattering[0, i0, 0, i1, 0, i2, 0] };
            ReadAccess  := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47246749696 + 1048576i0 + 2048i1 + 4i2 and o0↩
                  <= 47246749699 + 1048576i0 + 2048i1 + 4i2 };
            ReadAccess  := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47313862656 + 1048576i0 + 2048i1 + 4i2 and o0↩
                  <= 47313862659 + 1048576i0 + 2048i1 + 4i2 };
            WriteAccess := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47380975616 + 1048576i0 + 2048i1 + 4i2 and o0↩
                  <= 47380975619 + 1048576i0 + 2048i1 + 4i2 };
}
```

Анализ интервалов и режимов доступа к памяти после подстановки значений указателей и констант

KernelGen: точный анализ указателей

В KernelGen анализ указателей проводится во время исполнения, после подстановки значений:

sincos.c

```
void sincos(int nx, int ny, int nz, float* x, float* y, float* xy) {
    #pragma acc parallel
    for (int k = 0; k < nz; k++)
        for (int j = 0; j < ny; j++)
            for (int i = 0; i < nx; i++) {
                int idx = i + nx * j + nx * ny * k;
                xy[idx] = sin(x[idx]) + cos(y[idx]);
            }
}
```

result

```
Launching kernel __kernelgen_sincos_loop_10
    blockDim = { 32, 4, 4 }
    gridDim = { 16, 128, 16 }
Finishing kernel __kernelgen_sincos_loop_10
__kernelgen_sincos_loop_10 time = 2.300006e-02 sec
```

Другие полезные возможности

В системе портирования кода на GPU желательно обеспечить поддержку различных особенностей и стилей использования языков программирования, например:

- Распараллеливание циклов различных типов, не только арифметических for/do
- Распознавание неявных циклов (поэлементные операции с массивами в Fortran)
- Адресная арифметика (C/C++)
- …

Распараллеливание while-циклов

Благодаря низкоуровневому представлению LLVM IR и восстановлению из него структуры циклов в Polly, KernelGen способен, в частности, распараллеливать while- или goto-циклы *семантически эквивалентны* арифметическим for-/do-циклам (в OpenACC это не поддерживается):

```
  i = 1
  do while (i .le. nx)
    j = 1
    do while (j .le. nz)
      k = 1
      do while (k .le. ny)
        C(i, j) = C(i, j) + A(i, k) * B(k, j)
        k = k + 1
      enddo
      j = j + 1
    enddo
    i = i + 1
  enddo
```

```
Launching kernel __kernelgen_matmul__loop_9
    blockDim = { 32, 32, 1 }
    gridDim = { 2, 16, 1 }
Finishing kernel __kernelgen_matmul__loop_9
__kernelgen_matmul__loop_9 time = 0.00953514 sec
```

Распараллеливание неявных циклов

Краткая форма записи поэлементных операций с массивами (Fortran) преобразуется на уровне фронтенда в явные циклы, которые затем могут быть транслированы KernelGen в GPU-ядра:

```
1  program demo
2
3  implicit none
4  integer :: n
5  complex*16, allocatable, dimension(:) :: c1, c2,↩
       z
6  character(len=128) :: arg
7  integer :: i
8  real*8 :: v1, v2
9  real :: start, finish
10
11 call get_command_argument(1, arg)
12 read(arg, '(I64)') n
13
14 ! Allocate data arrays.
15 allocate(c1(n), c2(n), z(n))
16
17 ! Initialize arrays.
18 do i = 1, n
19   call random_number(v1)
20   call random_number(v2)
```

```
21   c1(i) = cmplx(v1, v2)
22   call random_number(v1)
23   call random_number(v2)
24   c2(i) = cmplx(v1, v2)
25 enddo
26
27 ! Implicit computational loop
28 call cpu_time(start)
29 z = conjg(c1) * c2
30 call cpu_time(finish)
31
32 write(*,*) 'compute time = ', finish - start
33
34 print *, 'z min = (', minval(realpart(z)), &
35   ', ', minval(imagpart(z)), '), max = (', &
36   maxval(realpart(z)), ', ', minval(imagpart(z)), ')'
37
38 ! Deallocate arrays.
39 deallocate(c1, c2, z)
40
41 end program demo
```

Распараллеливание неявных циклов

Пример компилируется как обычно, но с использованием *kernelgen-gfortran* вместо *gfortran*:

```
$ kernelgen-gfortran -O3 -std=c99 conjg.f90 -o conjg
```

```
$ ./conjg $((256*256*256))
 compute time =    0.10800600
 z min = (  7.18319034686704879E-006 ,  -0.99788337878880551      ), max = (  ↩
      1.9723582564715194      ,  -0.99788337878880551      )
```

```
$ kernelgen_runmode=1 ./conjg $((256*256*256))
 compute time =    2.80020237E-02
 z min = (  7.18319034686704879E-006 ,  -0.99788337878880551      ), max = (  ↩
      1.9723582564715194      ,  -0.99788337878880551      )
```

OpenACC: нет арифметики указателей

Компилятор не поддерживает распараллеливание циклов, содержащих арифметику указателей:

sincos.c

```
void sincos(int nx, int ny, int nz, float* x, float* y, float* xy) {
    float *xp = x, *yp = y, *xyp = xy;

    #pragma acc parallel
    for (int k = 0; k < nz; k++)
        for (int j = 0; j < ny; j++)
            for (int i = 0; i < nx; i++)
                *(xyp++) = sin(*(xp++)) + cos(*(yp++));
}
```

```
$ make
pgcc -fast -Minfo=accel -ta=nvidia,time -Mcuda=keepgpu,keepbin,keepptx,ptxinfo -c ../sincos.c -o sincos.o
PGC-W-0155-Pointer assignments are not supported in accelerator regions:  xyp (../sincos.c: 34)
PGC-W-0155-Accelerator region ignored  (../sincos.c: 29)
PGC/x86-64 Linux 13.2-0: compilation completed with warnings
```

KernelGen: есть арифметика указателей

Поскольку на уровне LLVM IR любой доступ к массивам выражается указателями и смещениями, индексируемые массивы и указатели поддерживаются одинаково хорошо:

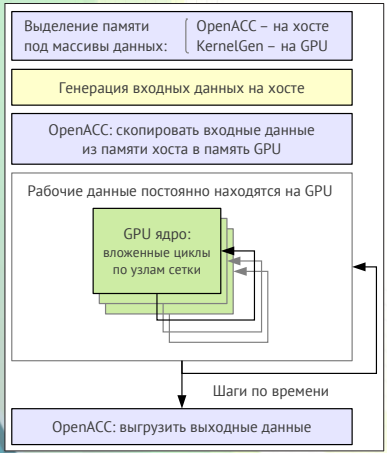
sincos.c

```
void sincos(int nx, int ny, int nz, float* x, float* y, float* xy) {
    float *xp = x, *yp = y, *xyp = xy;

    #pragma acc parallel
    for (int k = 0; k < nz; k++)
        for (int j = 0; j < ny; j++)
            for (int i = 0; i < nx; i++)
                *(xyp++) = sin(*(xp++)) + cos(*(yp++));
}
```

```
Launching kernel __kernelgen_sincos_loop_10
    blockDim = { 32, 4, 4 }
    gridDim = { 16, 128, 16 }
Finishing kernel __kernelgen_sincos_loop_10
__kernelgen_sincos_loop_10 time = 2.298868e-02 sec
```

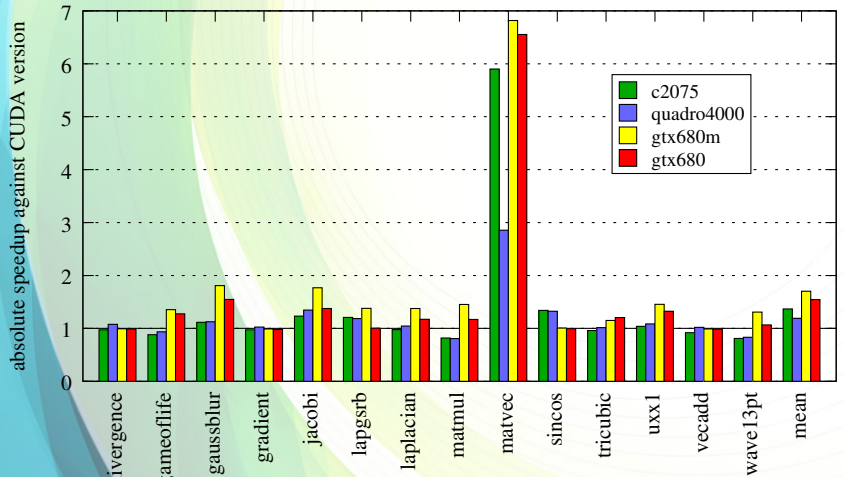
Тестирование производительности



Набор тестов сформирован из небольших программ со структурой, соответствующей типичной явной численной схеме:

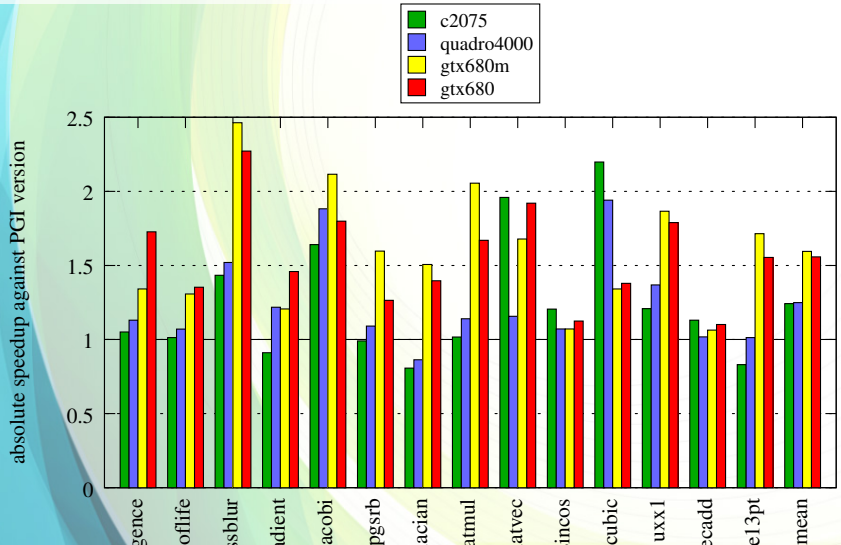
- Внешние итарации по времени (последовательные)
- Внутренние 2-х или 3-мерные циклы по узлам регулярной сетки (параллельные)
- Входные и выходные данные постоянно находятся в памяти GPU без промежуточных загрузок/выгрузок
- Корректность результатов оценивается контрольной суммой

KernelGen vs CUDA (вручную)



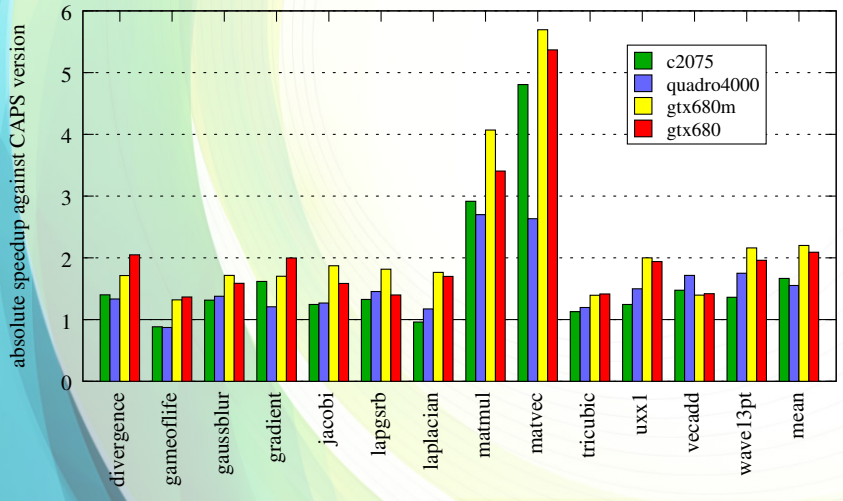
- Точность: **двойная**
- Компиляторы: KernelGen r1787, CUDA Toolkit v5.5
- GPU: NVIDIA Tesla C2075 (GF110, sm_20), NVIDIA GTX 680M (GK104, sm_30), NVIDIA GTX 680 (GK104, sm_30), NVIDIA Quadro 4000 (sm_20)
- Значения бóльшие 1 – ядро, собранное KernelGen быстрее ядра PGI, значения меньшие 1 – ядро PGI быстрее KernelGen (на том же самом GPU)
- Измерения осреднены по 10 тестам, по 10 итераций в каждом

KernelGen vs PGI OpenACC



- Точность: **двойная**
- Компиляторы: KernelGen r1787, PGI OpenACC 13.2
- GPU: NVIDIA Tesla C2075 (GF110, sm_20), NVIDIA GTX 680M (GK104, sm_30), NVIDIA GTX 680 (GK104, sm_30), NVIDIA Quadro 4000 (sm_20)
- Значения бóльшие 1 – ядро, собранное KernelGen быстрее ядра PGI, значения меньшие 1 – ядро PGI быстрее KernelGen (на том же самом GPU)
- Измерения осреднены по 10 тестам, по 10 итераций в каждом

KernelGen vs CAPS HMPP/OpenACC



- Точность: **двойная**
- Компиляторы: KernelGen r1787, CAPS HMPP 3.2.4
- GPU: NVIDIA Tesla C2075 (GF110, sm_20), NVIDIA GTX 680M (GK104, sm_30), NVIDIA GTX 680 (GK104, sm_30), NVIDIA Quadro 4000 (sm_20)
- Значения бóльшие 1 – ядро, собранное KernelGen быстрее ядра CAPS, значения меньшие 1 – ядро CAPS быстрее KernelGen (на том же самом GPU)
- Измерения осреднены по 10 тестам, по 10 итераций в каждом

Влияние режима кэширования

1. Поскольку текущие генераторы ядер явно не используют разделяемую память, в KernelGen конфигурация кэша установлена в «большой L1-кэш», что дает небольшой дополнительный прирост

```
// Since KernelGen does not utilize shared memory at the moment,
// use larger L1 cache by default.
CU_SAFE_CALL(cuCtxSetCacheConfig(CU_FUNC_CACHE_PREFER_L1));
```

Влияние вычислительного грида GPU

2 KernelGen использует блоки $\{128, 1, 1\}$ и 3-мерную сетку, когда как PGI и CAPS могут выбирать сетку по умолчанию или по gang/vector. Во всех тестах для PGI и CAPS установлены оптимальные gang/vector, иначе KernelGen сильно выигрывает.

Сетка по умолчанию PGI и сетка KernelGen, задача $512 \times 256 \times 256$, одинарная точность:

```
Accelerator Kernel Timing data
/home/marcusmae/forge/kernelgen/tests/perf/divergence/pgi/../divergence.c
  divergence  NVIDIA  devicenum=0
        time(us): 154,660
        62: kernel launched 10 times
            grid: [4x254]  block: [128]
             device time(us): total=154,660 max=15,560 min=15,373 avg=15,466
            elapsed time(us): total=154,742 max=15,570 min=15,381 avg=15,474
```

```
Kernel function call __kernelgen_divergence_loop_10
__kernelgen_divergence_loop_10 @ 0xeb32b61a9530d97f53f77c5abbd67132
Launching kernel __kernelgen_divergence_loop_10
    blockDim = { 128, 1, 1 }
    gridDim = { 4, 254, 254 }
Finishing kernel __kernelgen_divergence_loop_10
__kernelgen_divergence_loop_10 time = 7.912811e-03 sec
```

Влияние общей оптимизации кода

3 В KernelGen наблюдаются проблемы с оптимизацией редукции (matmul, matvec) и GPU-математики (sincos):

```
CUDA.LoopHeader.x.preheader:                      ; preds = %"Loop Function Root"
  %p_newGEPInst.cloned = getelementptr float* inttoptr (i64 47380979712 to float*)
  store float 0.000000e+00, float * %p_newGEPInst.cloned
  %p_.moved.to.4.cloned = shl nsw i64 %3, 9
  br label %polly.loop_body
CUDA.AfterLoop.x:                                 ; preds = %polly.loop_body, %"Loop Function Root"
  ret void
polly.loop_body:                                  ; preds = %polly.loop_body, %CUDA.LoopHeader.x.preheader
  %_p_scalar_ = phi float [ 0.000000e+00, %CUDA.LoopHeader.x.preheader ], [ %p_8, %polly.loop_body ]
  %polly.loopiv10 = phi i64 [ 0, %CUDA.LoopHeader.x.preheader ], [ %polly.next_loopiv, %polly.loop_body ]
  %polly.next_loopiv = add i64 %polly.loopiv10, 1
  %p_ = add i64 %polly.loopiv10, %p_.moved.to.4.cloned
  %p_newGEPInst9.cloned = getelementptr float* inttoptr (i64 47246749696 to float*), i64 %p_
  %p_newGEPInst12.cloned = getelementptr float* inttoptr (i64 47380971520 to float*), i64 %polly.loopiv10
  %_p_scalar_5 = load float* %p_newGEPInst9.cloned
  %_p_scalar_6 = load float* %p_newGEPInst12.cloned
  %p_7 = fmul float %_p_scalar_5, %_p_scalar_6
  %p_8 = fadd float %_p_scalar_, %p_7
  store float %p_8, float* %p_newGEPInst.cloned
  %exitcond = icmp eq i64 %polly.next_loopiv, 512
  br i1 %exitcond, label %CUDA.AfterLoop.x, label %polly.loop_body
```

Влияние общей оптимизации кода

3 В KernelGen наблюдаются проблемы с оптимизацией редукции (matmul, matvec) и GPU-математики (sincos):

```
CUDA.LoopHeader.x.preheader:                      ; preds = %"Loop Function Root"
  %p_newGEPInst.cloned = getelementptr float* inttoptr (i64 47380979712 to float*)
  store float 0.000000e+00, float * %p_newGEPInst.cloned
  %p_.moved.to.4.cloned = shl nsw i64 %3, 9
  br label %polly.loop_body
CUDA.AfterLoop.x:                                 ; preds = %polly.loop_body, %"Loop Function Root"
  ret void
polly.loop_body:                                  ; preds = %polly.loop_body, %CUDA.LoopHeader.x.preheader
  %_p_scalar_ = phi float [ 0.000000e+00, %CUDA.LoopHeader.x.preheader ], [ %p_8, %polly.loop_body ]
  %polly.loopiv10 = phi i64 [ 0, %CUDA.LoopHeader.x.preheader ], [ %polly.next_loopiv, %polly.loop_body ]
  %polly.next_loopiv = add i64 %polly.loopiv10, 1
  %p_ = add i64 %polly.loopiv10, %p_.moved.to.4.cloned
  %p_newGEPInst9.cloned = getelementptr float* inttoptr (i
  %p_newGEPInst12.cloned = getelementptr float* inttoptr (
  %_p_scalar_5 = load float* %p_newGEPInst9.cloned
  %_p_scalar_6 = load float* %p_newGEPInst12.cloned
  %p_7 = fmul float %_p_scalar_5, %_p_scalar_6
  %p_8 = fadd float %_p_scalar_, %p_7
  store float %p_8, float* %p_newGEPInst.cloned
  %exitcond = icmp eq i64 %polly.next_loopiv, 512
  br i1 %exitcond, label %CUDA.AfterLoop.x, label %polly.loop_body
```

Помечены входной, выходной блоки и тело цикла, так как они выглядят в LLVM IR

Влияние общей оптимизации кода

3 В KernelGen наблюдаются проблемы с оптимизацией редукции (matmul, matvec) и GPU-математики (sincos):

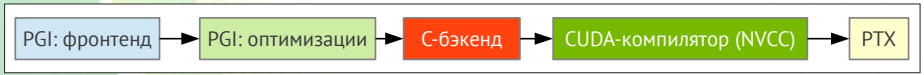
```
CUDA.LoopHeader.x.preheader:                      ; preds = %"Loop Function Root"
  %p_newGEPInst.cloned = getelementptr float* inttoptr (i64 47380979712 to float*)
  store float 0.000000e+00, float * %p_newGEPInst.cloned
  %p_.moved.to.4.cloned = shl nsw i64 %3, 9
  br label %polly.loop_body
CUDA.AfterLoop.x:                                 ; preds = %polly.loop_body, %"Loop Function Root"
  ret void
polly.loop_body:                                  ; pred
  %_p_scalar_ = phi float [ 0.000000e+00, %CUDA.LoopHead
  %polly.loopiv10 = phi i64 [ 0, %CUDA.LoopHeader.x.preh
  %polly.next_loopiv = add i64 %polly.loopiv10, 1
  %p_ = add i64 %polly.loopiv10, %p_.moved.to.4.cloned
  %p_newGEPInst9.cloned = getelementptr float* inttoptr
  %p_newGEPInst12.cloned = getelementptr float* inttoptr
  %_p_scalar_5 = load float* %p_newGEPInst9.cloned
  %_p_scalar_6 = load float* %p_newGEPInst12.cloned
  %p_7 = fmul float %_p_scalar_5, %_p_scalar_6
  %p_8 = fadd float %_p_scalar_, %p_7
  store float %p_8, float* %p_newGEPInst.cloned
  %exitcond = icmp eq i64 %polly.next_loopiv, 512
  br i1 %exitcond, label %CUDA.AfterLoop.x, label %polly.loop_body
```

Правильно оптимизированная редукция должна располагать аккумулятор на регистре и записывать его в память в конце один раз. Здесь запись в память происходит на каждой итерации!

Влияние генерации кода

4 PGI и CAPS – source-to-source компиляторы, когда как KernelGen – полный компилятор, благодаря LLVM NVPTX backend

PGI/CAPS:



KernelGen:

Анализ параллелизма

KernelGen анализирует зависимости данных и отображает параллельные циклы на нити и блоки GPU. Код последовательных циклов остается без изменений.

```
1  subroutine match_filter(HH, szhh, XX, szxx, YY, szyy)
2
3    implicit none
4    integer(kind=IKIND), intent(in) :: szhh, szxx, szyy
5    real(kind=RKIND), intent(in) :: HH(szhh), XX(szxx)
6    real(kind=RKIND), intent(out) :: YY(szyy)
7    integer(kind=IKIND) :: i, j
8    integer, parameter :: rkind = RKIND
9
10   ! This loop will be parallelized
11   do i = 1, szyy
12     YY(i) = 0.0_rkind
13     ! This loop will not be parallelized
14     do j = 1, szhh
15       YY(i) = YY(i) + XX(i + j - 1) * HH(j)
16     enddo
17   enddo
18
19 end subroutine match_filter
```

Анализ параллелизма

KernelGen анализирует зависимости данных и отображает параллельные циклы на нити и блоки GPU. Код последовательных циклов остается без изменений.

```
1  subroutine match_filter(HH, szhh, XX, szxx, YY, szyy)
2
3    implicit none
4    integer(kind=IKIND), intent(in) :: szhh, szxx, szyy
5    real(kind=RKIND), intent(in) :: HH(szhh), XX(szxx)
6    real(kind=RKIND), intent(out) :: YY(szyy)
7    integer(kind=IKIND) :: i, j
8    integer, parameter :: rkind = RKIND
9
10   ! This loop will be parallelized
11   do i = 1, szyy
12     YY(i) = 0.0_rkind
13     ! This loop will not be parallelized
14     do j = 1, szhh
15       YY(i) = YY(i) + XX(i + j - 1) * HH(j)
16     enddo
17   enddo
18
19 end subroutine match_filter
```

Распараллеливание редукции пока не поддерживается, этот цикл останется последовательным

Анализ параллелизма

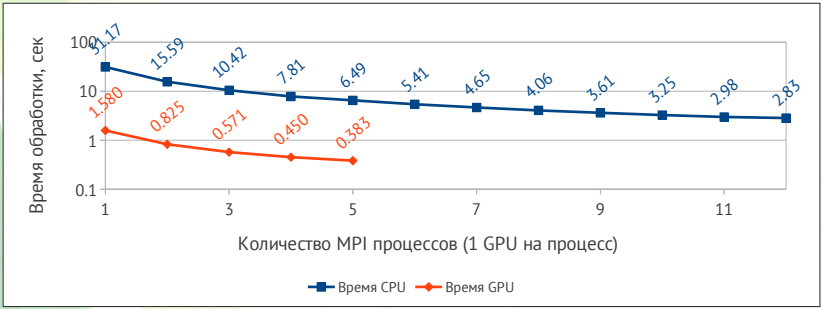
KernelGen анализирует зависимости данных и отображает параллельные циклы на нити и блоки GPU. Код последовательных циклов остается без изменений.

```
1  subroutine match_filter(HH, szhh, XX, szxx, YY, szyy)
2
3    implicit none
4    integer(kind=IKIND), intent(in) :: szhh, szxx, szyy
5    real(kind=RKIND), intent(in) :: HH(szhh), XX(szxx)
6    real(kind=RKIND), intent(out) :: YY(szyy)
7    integer(kind=IKIND) :: i, j
8    integer, parameter :: rkind = RKIND
9
10   ! This loop will be parallelized
11   do i = 1, szyy
12     YY(i) = 0.0_rkind
13     ! This loop will not be parallelized
14     do j = 1, szhh
15       YY(i) = YY(i) + XX(i + j - 1) * HH(j)
16     enddo
17   enddo
18
19 end subroutine match_filter
```

Однако, внешний цикл будет распознан как параллельный, и внутренний послед. цикл будет работать на каждой нити GPU

Взаимодействие с MPI

KernelGen естественным образом взаимодействует с MPI, отображая множество MPI-процессов на множество GPU:



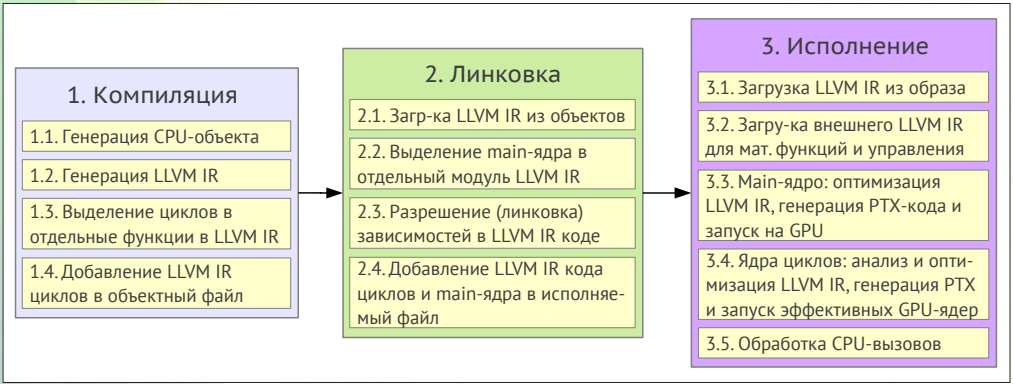
Сравнение производительности корреляционного фильтра, скомпилированного KernelGen r1787 для 1-12 CPU-ядер Intel Xeon X5670 и 1-5 GPU NVIDIA Tesla C2070 GPUs (Fermi sm_20)

Зависимости KernelGen

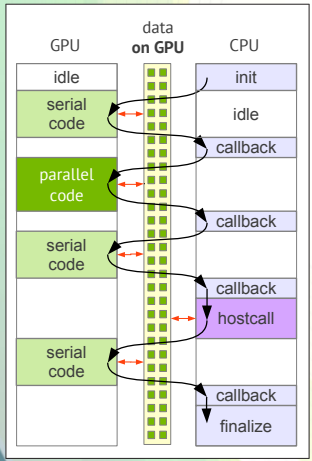
- **GCC** – фронтенд и элементы линковщика (GPL)
- **DragonEgg** – плагин для GCC, преобразующий GIMPLE IR в LLVM IR (GPL)
- **LLVM** – основная инфраструктура компилятора (BSD)
- **Polly** – анализ параллельности циклов (BSD+GPL)
- **NVPTX backend** – генерация PTX-псевдоассемблера или LLVM IR (BSD)
- **PTXAS** – генерация GPU ISA (CUBIN) из PTX-кода (исходный код недоступен)
- **AsFermi** – модификации кода на уровне Fermi GPU ISA (CUBIN) (MIT/BSD)
- **NVIDIA GPU driver** – исполнение сгенерированных ядер на GPU (исходный код недоступен)

Организация компилятора KernelGen

KernelGen основан на элементах компилятора GCC, расширяя его преобразованиями LLVM, которые активируются в зависимости от режима компиляции/исполнения.



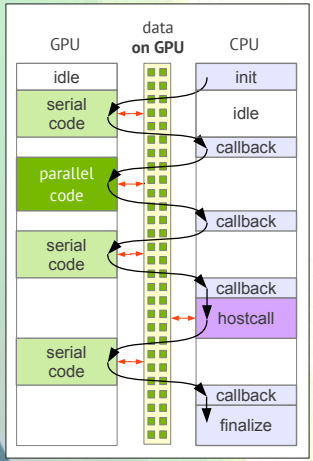
GPU-активное исполнение



KernelGen использует GPU-активную модель исполнения:

- Все действия инициируются с GPU (CPU «пассивен»)
- Все данные приложения (или MPI-узла) находятся в памяти GPU
- Операции с данными и управление исполнением производятся *main-ядром*, которое постоянно работает на GPU, в течение всего времени жизни приложения

GPU-активное исполнение

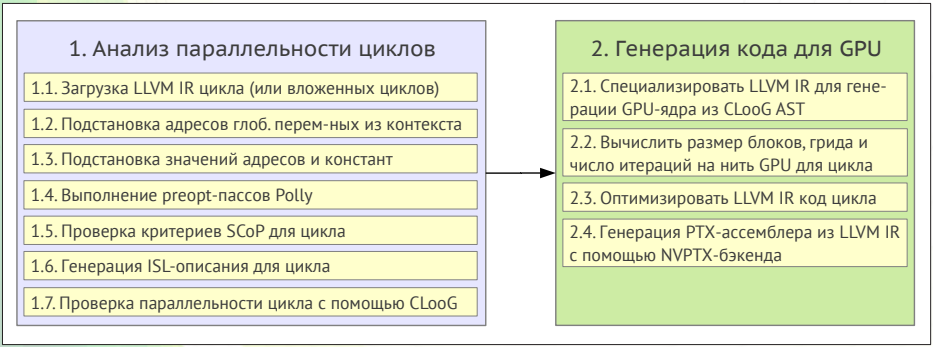


Преимущества:

- Нет необходимости явно производить пересылку используемых данных между хостом и GPU и находить скрытые зависимости (side-effects)
- Возможность запускать длинные непрерывные ядра на GPU, поддерживающих динамический параллелизм
- Легкое превращение CPU-MPI-узлов в GPU-MPI-узлы, возможно с CUDA-aware MPI
- GPU-функции изначально загруженные main-ядром также используются другими вычислительными ядрами, что уменьшает размер кода и время компиляции (в отличие от OpenACC-ядер, которые всегда содержат копии кода всех зависимостей)

Этапы анализа циклов в KernelGen

KernelGen проводит заключительную фазу анализа циклов и компиляцию ядер во время исполнения, привлекая доступную информацию о контексте использования кода. Если один и тот же цикл вызывается множество раз, то накладные расходы на этот процесс невелики.



Краткая схема генерации кода

Рассмотрим этапы генерации кода в KernelGen на примере следующей функции:

sincos.c

```
void sincos(int nx, int ny, int nz, float* x, float* y, float* xy) {
    for (int k = 0; k < nz; k++)
        for (int j = 0; j < ny; j++)
            for (int i = 0; i < nx; i++) {
                int idx = i + nx * j + nx * ny * k;
                xy[idx] = sin(x[idx]) + cos(y[idx]);
            }
}
```

Краткая схема генерации кода

Первым этапом является преобразование языка высокого уровня во внутреннее представление (GIMPLE IR) с помощью фронтенда компилятора GCC:

```
sincos (integer(kind=4) & restrict nx, integer(kind=4) & restrict ny, integer(kind=4) & restrict nz, real(kind=4)[0:D↩
      .1629] * restrict x, real(kind=4)[0:D.1626] * restrict y, real(kind=4)[0:D.1632] * restrict xy)
{
  ...
  D.1646 = ~stride.16;
  offset.19 = D.1646 - stride.18;
  {
    ...
    {
      ...
      {
        if (j <= D.1568) goto <D.1650>; else goto <D.1651>;
        <D.1650>:
        <D.1652>:
        {
          logical(kind=4) D.1576;
          {
            ...
            {
              logical(kind=4) D.1575;
              ...
              D.1676 = sincos_ijk (D.1675, D.1669);
              *xy[D.1663] = D.1676;
```

Краткая схема генерации кода

Затем плагином DragonEgg представление GIMPLE IR преобразуется в LLVM IR:

```
; ModuleID = '../sincos.f90'
target datalayout = "e-p:64:64:64-S128-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f16:16:16-f32:32:32-f64:64:64-f128
      :128:128-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64"
target triple = "x86_64-unknown-linux-gnu"

declare float @sincos_ijk_(...)
...
define void @sincos_(i32* noalias %nx, i32* noalias %ny, i32* noalias %nz, [0 x float]* noalias %x, [0 x float]* noalias
      %y, [0 x float]* noalias %xy) nounwind uwtable {
  ...
  %92 = add i64 %89, %91
  %93 = add i64 %87, %92
  %94 = add i64 %93, %41
  %95 = bitcast [0 x float]* %4 to float*
  %96 = getelementptr float* %95, i64 %94
  %97 = call float bitcast (float (...)* @sincos_ijk_ to float (float*, float*)*)(float* %96, float* %86) nounwind
  %98 = bitcast [0 x float]* %5 to float*
  %99 = getelementptr float* %98, i64 %76
  store float %97, float* %99, align 4
  %100 = icmp eq i32 %68, %66
  %101 = add i32 %68, 1
  %102 = icmp ne i1 %100, false
  br i1 %102, label %"7", label %"6"
```

Краткая схема генерации кода

На этапе компиляции KernelGen выделяет *потенциально* параллельные циклы в новые функции. Группы вложенных циклов выделяются рекурсивно, на случай если не все из них параллельны. Во время исполнения в LLVM IR подставляются значения констант и указателей, что облегчает распараллеливание:

```
; ModuleID = '__kernelgen_sincos__loop_3_module'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v16:16:16-v32:32:32-v64
      :64:64-v128:128:128-n16:32:64"
target triple = "nvptx64-unknown-unknown"
define void @__kernelgen_sincos__loop_3(i32*) nounwind {
"Loop Function Root":
  br label %"4.preheader.cloned"
  ...
  %8 = getelementptr [0 x float]* inttoptr (i64 47313862656 to [0 x float]*), i64 0, i64 %7
  %9 = getelementptr [0 x float]* inttoptr (i64 47246749696 to [0 x float]*), i64 0, i64 %7
  %10 = load float* %9, align 4
  %11 = call float @sinf(float %10) nounwind readnone alignstack(1)
  %12 = load float* %8, align 4
  %13 = call float @cosf(float %12) nounwind readnone alignstack(1)
  %14 = fadd float %11, %13
  %15 = getelementptr [0 x float]* inttoptr (i64 47380975616 to [0 x float]*), i64 0, i64 %7
  store float %14, float* %15, align 4
  ...
declare float @sinf(float) nounwind readnone alignstack(1)
declare float @cosf(float) nounwind readnone alignstack(1)
```

Краткая схема генерации кода

KernelGen анализирует независимость итераций цикла. В случае если цикл параллельный, LLVM IR специализируется для NVPTX (GPU-интринсики *@llvm.nvvm.**, атрибуты *ptx_kernel* and *ptx_device*):

```
; ModuleID = '__kernelgen_sincos__loop_3_module'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v16:16:16-v32:32:32-v64
      :64:64-v128:128:128-n16:32:64"
target triple = "nvptx64-unknown-unknown"

@__kernelgen_version = constant [15 x i8] c"0.2/1654:1675M\00"

define ptx_kernel void @__kernelgen_sincos__loop_3(i32* nocapture) nounwind alwaysinline {
"Loop Function Root":
  %tid.z = tail call ptx_device i32 @llvm.nvvm.read.ptx.sreg.tid.z()
  %ctaid.z = tail call ptx_device i32 @llvm.nvvm.read.ptx.sreg.ctaid.z()
  %PositionOfBlockInGrid.z = shl i32 %ctaid.z, 2
  %BlockLB.Add.ThreadPosInBlock.z = add i32 %PositionOfBlockInGrid.z, %tid.z
  ...
  %p_28 = tail call ptx_device float @sinf(float %_p_scalar_) nounwind readnone alignstack(1)
  %_p_scalar_29 = load float* %p_scevgep6
  %p_30 = tail call ptx_device float @cosf(float %_p_scalar_29) nounwind readnone alignstack(1)
  %p_31 = fadd float %p_28, %p_30
  store float %p_31, float* %p_scevgep
  br label %CUDA.AfterLoop.z
  ...
```

Краткая схема генерации кода

С помощью LLVM NVPTX-бекенда из LLVM IR генерируется PTX-ассемблер:

```
.visible .entry __kernelgen_sincos__loop_3(.param .u64 __kernelgen_sincos__loop_3_param_0)
{
    ...
// BB#0:                                // %Loop Function Root
    mov.u32  %r0, %tid.z;
    mov.u32  %r1, %ctaid.z;
    shl.b32  %r1, %r1, 2;
    add.s32  %r2, %r1, %r0;
    @%p0 bra  BB0_4;
    ...
    // Callseq Start 42
    {
    .reg .b32 temp_param_reg;
    // <end>}
    .param .b32 param0;
    st.param.f32 [param0+0], %f0;
    .param .b32 retval0;
    call.uni (retval0),
    sinf,
    (
    param0
    );
    ...
```

Краткая схема генерации кода

Наконец, для функции генерируется Fermi ISA ассемблер в режиме no-cloning. В пустые вызовы внешних функций (*JCAL 0x0*) подставляются актуальные адреса функций, ранее загруженных main-ядром.

```
Function : __kernelgen_sincos__loop_3
/*0008*/     /*0x10005de428004001*/  MOV R1, c [0x0] [0x44];
/*0010*/     /*0x9c00dc042c000000*/  S2R R3, SR_CTAid_Z;
/*0018*/     /*0x8c001c042c000000*/  S2R R0, SR_Tid_Z;
...
/*0120*/     /*0x08451c036000c000*/  SHL R20, R4, 0x2;
/*0128*/     /*0x14055c4340000000*/  ISCADD R21, R0, R5, 0x2;
/*0130*/     /*0x01411c020c0080c0*/  IADD32I R4.CC, R20, 0x203000;
/*0138*/     /*0x2d515c434800c000*/  IADD.X R5, R21, 0xb;
/*0148*/     /*0x00411c8584000000*/  LD.E R4, [R4];
/*0150*/     /*0x00010007100017b2*/  JCAL 0x5ec80;
/*0158*/     /*0x01419c020c108100*/  IADD32I R6.CC, R20, 0x4204000;
/*0160*/     /*0x10009de428000000*/  MOV R2, R4;
/*0168*/     /*0x2d51dc434800c000*/  IADD.X R7, R21, 0xb;
/*0170*/     /*0x00611c8584000000*/  LD.E R4, [R6];
/*0178*/     /*0x00010007100018eb*/  JCAL 0x63ac0;
/*0188*/     /*0x01419c020c208140*/  IADD32I R6.CC, R20, 0x8205000;
/*0190*/     /*0x10201c0050000000*/  FADD R0, R2, R4;
/*0198*/     /*0x2d51dc434800c000*/  IADD.X R7, R21, 0xb;
/*01a0*/     /*0x00601c8594000000*/  ST.E [R6], R0;
/*01a8*/     /*0x00001de780000000*/  EXIT;
```

Ограничения KernelGen

KernelGen всё ещё имеет ряд ограничений:

- Поддерживается только единичная индексация в циклах
- Так же как и в OpenACC, не поддерживается динамическая косвенная индексация (например, $a[b[i]]$)
- Не поддерживается распараллеливание редукции (в OpenACC есть соотв. директива)
- Polly не всегда распараллеливает код из-за недостаточной или избыточной оптимизации

Выводы: применимость

1. В рамках проекта KernelGen реализован прототип полного компилятора, автоматически генерирующего GPU-код для оригинального CPU-приложения
2. Значительное внимание уделено поддержке различных возможностей языков C и Fortran
3. Производительность ядер, генерируемых KernelGen находится на уровне современных коммерческих компиляторов OpenACC (PGI и CAPS), а в некоторых случаях – в несколько раз выше
4. KernelGen облегчает портирование приложений, поскольку в нём отсутствуют многие ограничения, присутствующие в OpenACC
5. Центрированная на GPU модель исполнения позволяет легко переводить CPU-MPI-приложение в GPU-MPI-приложение
6. KernelGen основан на бесплатных и большей частью открытых технологиях

Поиск новых оптимизаций

1. Помогает ли тайлинг на современных GPU?
2. Есть ли какие-то другие полезные оптимизации?

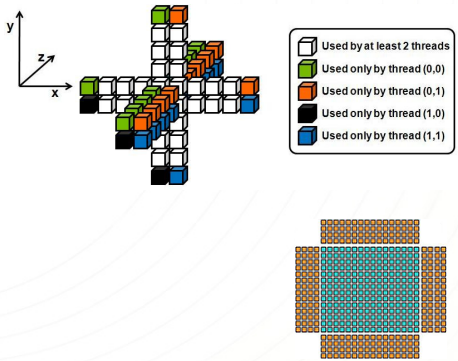
Ускорение на 13% при использовании выравнивания:

```
for (int k = 2+ k_start; k < ns — 2; k += k_inc)
    for (int j = 2 + j_start; j < ny — 2; j += j_inc)
        for (int i = i_start; i < nx — 2; i += i_inc)
        {
            if (i < 2) continue;
            ...
        }
```

3D Finite Difference Computation on GPUs using CUDA

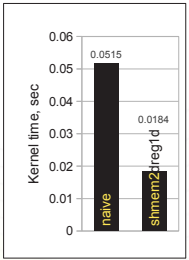
Популярная статья Paulius Micikevicius, **298** цитирований:

- Численная схема для 3-мерного волнового уравнения (6-12 порядка по пр-вку, 2 порядка по времени)
- 2-мерные слои кешируются в shared-памяти GPU
- Колонки третьей размерности кешируются в регистрах потока
- Опубликовано в 2009 г., тестировалось на Tesla S1060 (GT200)



Воспроизведение результата на S1070

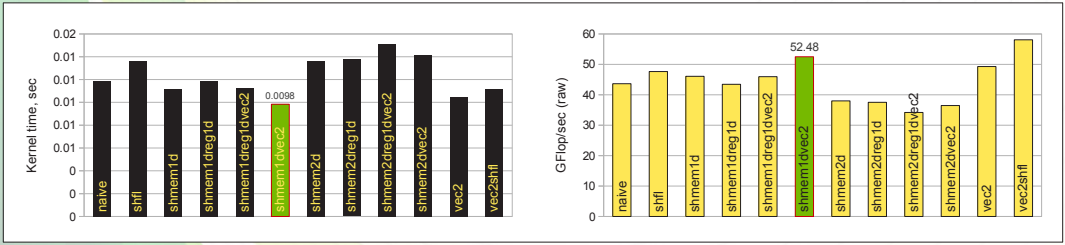
- $\{32, 16, 1\}$ потоков в блоке, maxrregcount=32
- каждый поток обрабатывает точки одной вертикальной колонки
- в вертикальной колонке точки кешируются в регистрах
- *naive* – «наивная» CUDA-версия, *shmem2dreg1d* – с shared-памятью и кешем в регистрах
- *shmem2dreg1d* почти в **3 раза** быстрее *naive* на S1070 – запомним этот результатат!



`wave13pt` на Tesla S1070 (GT200/SM_13), одинарная точность

Лучшая оптимизация вручную

- 1-мерная shared-память с векторизацией – самый быстрый вариант на GTX 680M
- вклад shared-памяти незначителен
- версия только с векторизацией – самый быстрый вариант на Tesla K20
- по сравнению с 3-кратным приростом на S1070, здесь shared-память почти бесполезна!



wave13pt на GeForce GTX 680M (GK104/SM_30), одинарная точность

Профилирование векторизации

- Код, сгенерированный для memory-bound алгоритма оказался compute-bound
- 2-элементная векторизация улучшает эффективность памяти и уменьшает долю арифметики (меньше индексации?)

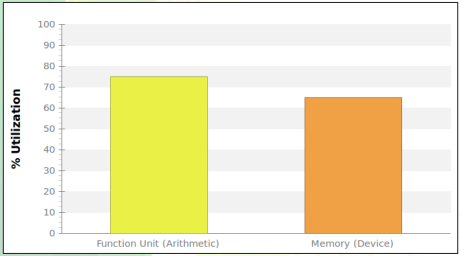


Figure: «Наивная» CUDA-реализация

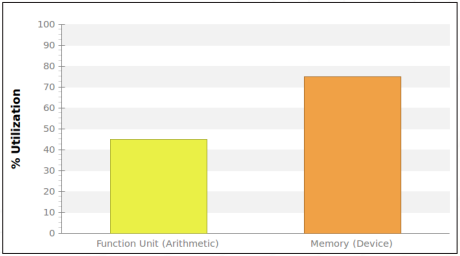


Figure: Векторизованная CUDA-реализация

Профилирование векторизации

- Пропускная способность памяти выше в векторизованной версии (очень близка к *cuMemcpyDtoD*, кот. даёт 84 Гб/сек на данном устройстве).

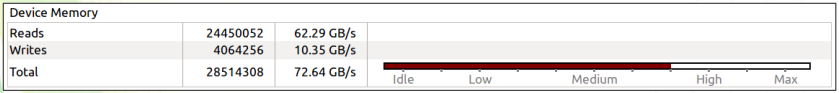


Figure: «Наивная» CUDA-реализация

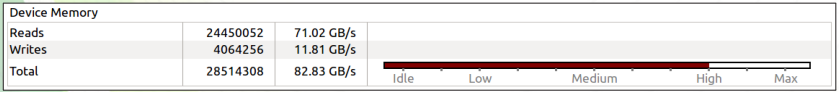


Figure: Векторизованная CUDA-реализация

Выводы: дальнейшая оптимизация

1. Наиболее часто цитируемая оптимизация – тайлинг с shared-памятью – почти не даёт прироста на совр. GPU с L1/L2-кешами $\Rightarrow$ нет смысла реализовывать в компиляторе
2. Векторизация LD/ST даёт 10-15%-ускорение и по-видимому малоизвестна (упоминается 1 раз в блоге) $\Rightarrow$ имеет смысл реализовать в компиляторе, но сложно объединять с polyhedral analysis

CUDA 6.0’ Unified Memory

CPU-код

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA-код с поддержкой Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
   cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

$\Rightarrow$ что, если подобную замену сделать на уровне Glibc?

KernelGen

Ссылка на данную презентацию:
http://kernelgen.org/msu_sct/

Рассылка проекта:
kernelgen-devel@lists.hpcforge.org

Спасибо!

HP2C

Università della Svizzera italiana
Facoltà di scienze informatiche

APC LLC