1.5. Использование современных архитектур для обработки данных в режиме реального времени.

Параллельные вычисления: сравнение технологий OpenMP и MPI. Альтернативные аппаратные решения.
Использование графических ускорителей для технологических расчетов.

Адаптация под различные платформы

В соответствие с приведенной выше математической моделью, расчет движения волны происходит в два этапа: на первом шаге производится вычисление смещения волны вдоль оси X, на втором – вдоль оси Y. Используется подход расщепления направлений. При этом расчет i-ой и j-ой строки вдоль оси X можно производить независимо. Аналогичная ситуация и для расчета вдоль оси Y. Основной цикл схематично можно представить следующим образом:

```
[для каждого временного шага]

  // расчет вдоль оси X
  [для каждого столбца поля решения - i]
    [вычислить инварианты для столбца i]
    [произвести расчет – функция swater]
    [записать результаты расчета — перевести
         инварианты в исходные значения]
  // расчет вдоль оси Y
  [для каждого ряда поля решения - i]
    [вычислить инварианты для для ряда i]
    [произвести расчет – функция swater]
    [записать результаты расчета — перевести
         инварианты в исходные значения]

   [записать решение на соответствующем временном шаге]
```

Для расчета используются четыре матрицы: глубина (**d**), высота волны (**q**), скорость движения волны (**u**) вдоль оси X и скорость движения волны (**v**) вдоль оси Y. Размерность матриц равна размерности расчетного поля и для акватории Тихого океана составляет 2581x2879.

В целях оптимизации в последовательной программе вторая половина итерации по времени обрамлена функциями транспонирования матриц **q**, **u** и **v**. Матрица **d** (глубина) остается неизменной и может быть транспонирована один раз перед основным циклом.

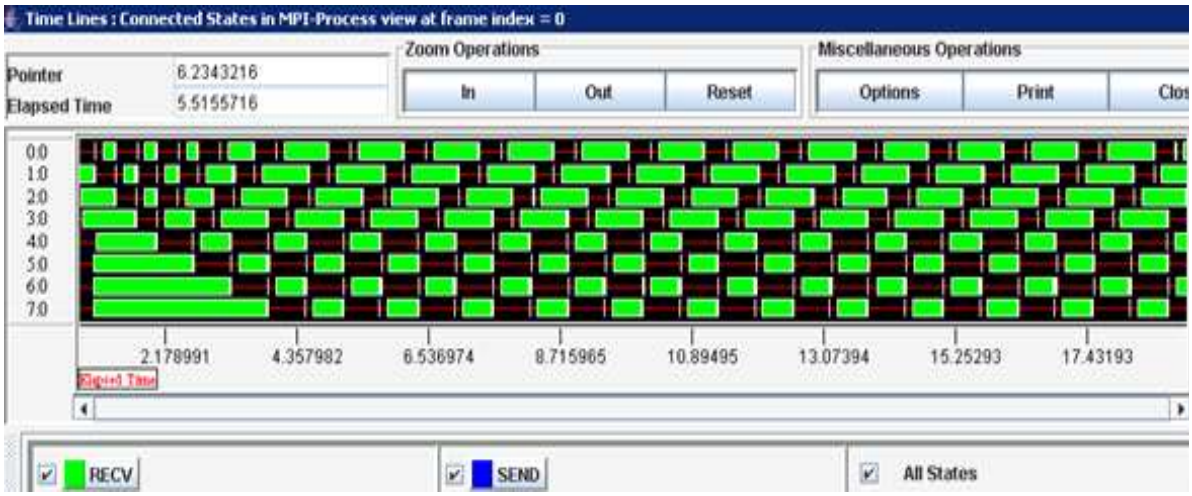
Основной цикл расчета на языке Си представлен на рис 1.

```
for(n=0; n<mmax; n++){
   for(j = 0; j < n1; j ++) {  // calculate along Y
      for(i=0; i<n2; i++) {
         qw[i] = u[j*n2 + i] - 2.0f * sqrtf( 9.8f*q[j*n2 + i]);
         uw[i] = u[j*n2 + i] + 2.0f * sqrtf( 9.8f*q[j*n2 + i]);
         vw[i] = v[j*n2 + i];
      } //for i
     swater(uw, qw, vw, &d[j*n2], u1, q1, v1, &n2, h2, &grnd, &t);

      for(i=0; i<n2; i++){
```

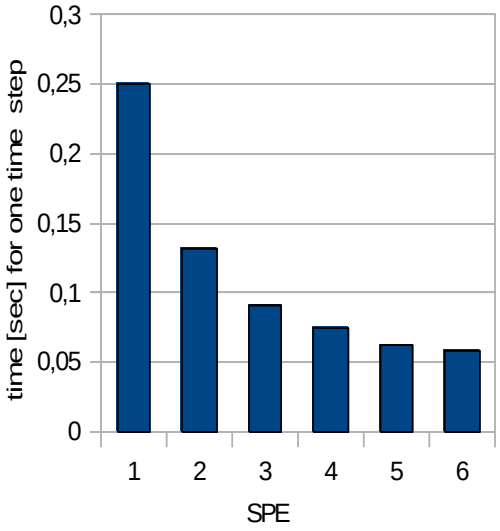
```
            q[j*n2 + i] = (u1[i] - q1[i]) * (u1[i] - q1[i]) /(9.8f*16.0f);
            u[j*n2 + i] = (u1[i] + q1[i]) * .5f;
           v[j*n2 + i] = v1[i];
        } //for i
    } //  for j

    transpose(q, n2, n1); transpose(u, n2, n1); transpose(v, n2, n1);

   for(j=0; j<n2; j++){  // calculate along X
        for(i=0; i<n1; i++){
            qw[i] = v[j*n1 + i] - 2.0f * sqrtf( 9.8f*q[j*n1 + i]);
            uw[i] = v[j*n1 + i] + 2.0f * sqrtf( 9.8f*q[j*n1 + i]);
            vw[i] = u[j*n1 + i];
        } //for i

        swater(uw, qw, vw, &dt[j*n1], u1, q1, v1, &n1, h1, &grnd, &t);

        for(i=0; i<n1; i++){
            q[j*n1 + i] = (u1[i] - q1[i]) * (u1[i] - q1[i]) /(9.8f*16.0f);
           v[j*n1 + i] = (u1[i] + q1[i]) * .5;
            u[j*n1 + i] = v1[i];
        } //for i
    } //  for j

    transpose(q, n1, n2);   transpose(u, n1, n2);  transpose(v, n1, n2);
    // getting data from "sensors" and save result
}
```

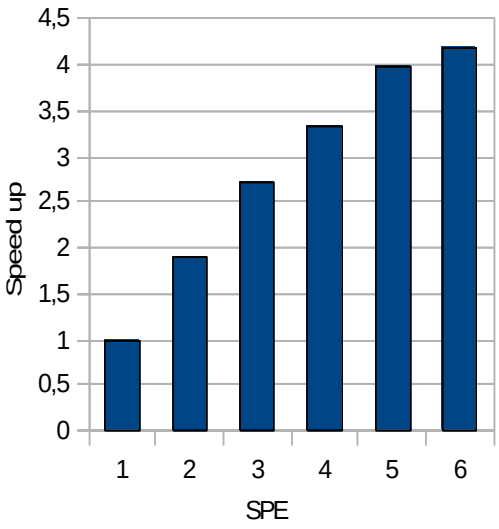
Рисунок 1: Основной цикл расчета последовательной программы на языке Си

Количество итераций основного цикла может составлять около 9000, что соответствует около 24 часам реального времени, которое требуется для распространения волны по всему океану. Все итерации содержат приблизительно равное количество операций и поэтому показателем ускорения может служить время выполнения одной итерации. Так для последовательной версии программы время выполнения одного шага по времени на процессоре Dual-Core AMD Opteron™ Processor 2218MHz составляет около 3.5 секунды. OpenMP версия программы на 4 ядрах одну итерацию рассчитывает около 1 секунды, на 8 ядрах — 0.53 секунды. Расчеты также выполнялись на кластере (MPI), но в виду того, что алгоритм требовал частого обмена данными между узлами кластера, большого ускорения получить не удалось — максимум 4 раза на 12 узлах.



Наиболее хорошие результаты удалось достичь на платформе SONY PlayStation3 (процессор IBM CELL BE, использование 6 вычислительных ядер). После проведенных оптимизаций получить 50-ти кратное ускорение по сравнению с исходной последовательной программой.





Вычисление вдоль оси X, а затем вдоль оси Y может производиться независимо для каждой строки/столбца. Более того, можно независимо вычислять значения инвариантов и новых значений высоты волны и скоростей вдоль осей для каждой точки пространства. Это дает основание рассчитывать, что данный алгоритм будет эффективен на архитектуре GPU.

Адаптация алгоритма под GPU

Один из стандартных подходов к адаптации программ под GPU — это последовательный перенос участков кода на GPU. В рамках реализации этого подхода был предложен следующий код:

```
for(int i=0; i<cfg.steps; i++){
    // calculate along X
    Invariants_X<<<dimGrid, dimBlock>>>(... , x_size, y_size);
    SWater_<<<dimGrid, dimBlock>>>(... , x_size, y_size);
    RInvariants_X<<<dimGrid, dimBlock>>>(... , x_size, y_size);

    // calculate along Y
    transpose<<<dimGrid, dimBlock>>>(d_qwdata, d_q1data, x_size, y_size);
    transpose<<<dimGrid, dimBlock>>>(d_uwdata, d_u1data, x_size, y_size);
    transpose<<<dimGrid, dimBlock>>>(d_vwdata, d_v1data, x_size, y_size);

    Invariants_Y<<<dimGrid_, dimBlock>>>(... , y_size, x_size);
    SWater_<<<dimGrid_, dimBlock>>>(..., y_size, x_size);
    RInvariants_Y<<<dimGrid_, dimBlock>>>(..., y_size, x_size);

    transpose<<<dimGrid_, dimBlock>>>(d_qwdata, d_q1data, y_size, x_size);
    transpose<<<dimGrid_, dimBlock>>>(d_uwdata, d_u1data, y_size, x_size);
    transpose<<<dimGrid_, dimBlock>>>(d_vwdata, d_v1data, y_size, x_size);
```

```
    // getting data from "sensors" and save result
    ...
}
```

Рисунок 5: Основной цикл расчета, перенесенный на CUDA

Для простоты был выбран размер блока потоков (ThreadBlock) равный 16x16 потоков. Таким образом, все пространство моделирования было разбито на равные блоки. Как в дальнейшем показало профилирование и эксперименты именно такая конфигурация является оптимальной для загрузки потоковых мультипроцессоров (Stream Multiprocessor). При этом на видеокарте NVIDIA GeForce 9800 GX2 время выполнения одной итерации составило в среднем 0.25 секунды. Оптимизация не выполнялась. При этом около 43% времени отнимало ядро SWater_, ядро transpose() занимало еще 26%.

Функции вычисления инвариантов и обратных инвариантов являются тривиальными и не нуждаются в комментариях. Реализация функции transpose() была взята из CUDA SDK. Наиболее сложной в плане реализации оказалась функция SWater_. Внутри нее требовалось проводить большое количество проверок на граничные условия (наличие берегов материков и островов), и вести интенсивное чтение данных из памяти. Для упрощения этой функции был заранее выполнен просчет некоторых условий, что позволило исключить вложенные проверки. Кроме того, все необходимые для расчета данные копировались в разделяемую память, после чего работа велась именно с ней.

При моделировании выяснилось, что вычисление инвариантов (ядра Invariants_X и Invariants_X) происходит с некоторой неточностью, что определяется наличием в ядрах вызова функции вычисления квадратного корня, которая дает ошибку в 1-2 младших битах, что в конечном итоге приводило к самопроизвольному раскачиванию поверхности океана уже на 50 итерациях. Было рассмотрено несколько решений: перейти на вычисления инвариантов в double, использовать представление double с помощью двух float [7], и как вариант вообще избавиться от вычисления квадратного корня внутри основного цикла. Первые два варианта не приводили к большому провалу по производительности. На текущий момент времени каждый потоковый мультипроцессор содержит по 8 модулей для работы с числами с плавающей точкой с одинарной точностью и по 2 для работы с числами с двойной точностью. Последний вариант — отказ от вычисления квадратного корня — оказался наиболее простым и подходящим в плане реализации.

Для того, чтобы избавиться от постоянного вычисления квадратного корня было решено производить все вычисления в инвариантах и только на стадии сохранения данных моделирования производить пересчет инвариантов назад к значениям высоты волны и ее скоростям по направлениям. В результате код был модифицирован, поверхность океана перестала «раскачиваться» и время вычисления одной итерации уменьшилось и составило в среднем 0.23 сек на одну итерацию по времени. Еще на 10% удалось ускорить программу, заменив в ядре SWater_ некоторые операции деления на умножение. В соответствие с документацией деление занимает в 9 раз больше тактов, чем умножение: 36 против 4 тактов. Тем самым время вычисления одной итерации составило 0.21 секунду.

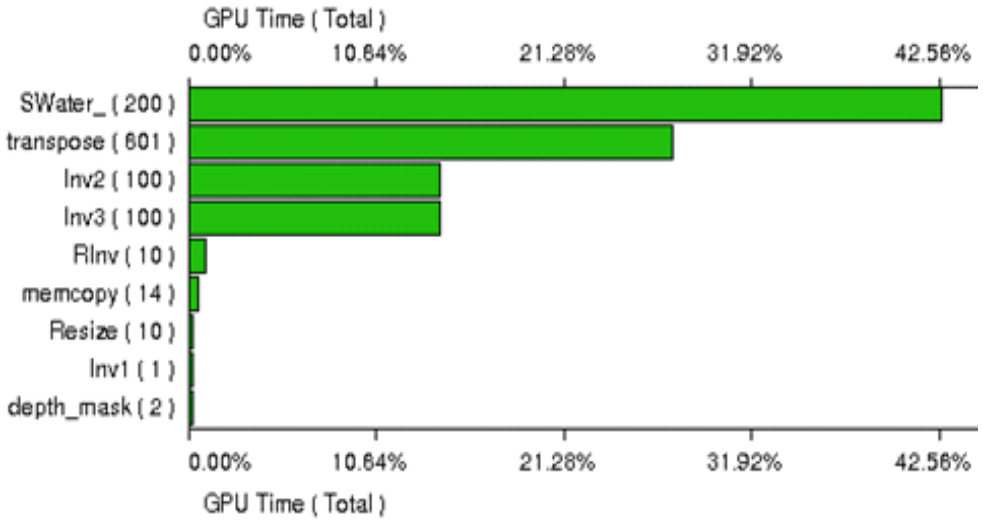


Рисунок 6: Распределение функций по занимаемому времени

Были проверены несколько путей оптимизации программы:

1. Сделать две функции SWater_, которые будут производит вычисления вдоль разных направлений, и тем самым избавиться от вызова функции transpose();
2. Выровнять начала строк всех матриц в памяти GPU для того, чтобы получить последовательно чтение. Заменить вызовы функций cudaMalloc() на cudaMallocPitch();
3. Перейти к работе с памятью через текстуры.

Как видно из вышеприведенного графика функция transpose занимает около 26% времени выполнения программы. В тоже время она используется только для того, чтобы создать удобное расположение данных в памяти для функции SWater_ на второй половине итерации. Это было оправдано для последовательной версии программы и программы на OpenMP, но является лишним для реализации на CUDA с учетом дальнейших оптимизаций. Таким образом была создана функция SWater_r, которая выполняла те же вычисления, что и функция SWater_, но вдоль другой оси. В результате время выполнения одного цикла по времени сократилось до 0.135 секунд. При этом упростилась и сама программа.

```
Inv1<<<dimGrid, dimBlock>>>(... , x_size, y_size);

for(int i=0; i<cfg.steps; i++){
   // calculate along X
   SWater_<<<dimGrid, dimBlock>>>(..., x_size, y_size);

   Inv2<<<dimGrid, dimBlock>>>(..., x_size, y_size);

   // calculate along Y
   SWater_r<<<dimGrid, dimBlock>>>(..., x_size, y_size);
   // getting data from "sensors" and save result
   ...

   Inv3<<<dimGrid, dimBlock>>>(..., x_size, y_size);
}// for cfg.steps
```

Рисунок 7: Основной цикл расчета после оптимизации

Для выявления направлений дальнейшей оптимизации использовался профилировщик (запуск программы с установленными переменными окружения CUDA_PROFILE и CUDA_PROFILE_CONFIG, или через утилиту cudaprof). Анализ показал, что несмотря на простой вид функции Inv2 и Inv3, непоследовательное чтение и запись происходят по 2.6 миллиона раз, в то время как последовательное — всего несколько тысяч.

```
__global__ void Inv2(float *q, float *u, float *v,
                     float *qw, float *uw, float *vw, int width, int height){
    unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int yIndex = blockIdx.y * blockDim.y + threadIdx.y;

    unsigned int indx = yIndex * width + xIndex;
    float bq, bu, bv;

    if((xIndex<width) && (yIndex<height)){
        bv = v[indx];
        bq = q[indx]*0.5f;
        bu = u[indx]*0.5f;

        qw[indx] = bv - (bu - bq);
        uw[indx] = bv + (bu - bq);
        vw[indx] = bu + bq;
    }
}
```

Рисунок 8: Ядро для пересчета инвариантов

Таблица 1: Количество последовательных/непоследовательных операций чтения/записи.

method	gld_incoherent	gld_coherent	gst_incoherent	gst_coherent
SWater_	4.92214e+06	105505	5.24688e+06	48144
Inv2	2.62344e+06	10935	5.24688e+06	43740
SWater_r	4.72e+06	91004	5.24688e+06	47772
RInv	2.62344e+06	10935	1.74896e+06	14580
Inv3	2.60116e+06	10935	5.20233e+06	43740

Величина в 2.6 миллиона непоследовательных операций чтения (gld_incoherent) — это приблизительно количество элементов в массивах qw, uw и vw из примера на рис 8. Таким образом, все элементы являются выравненными на границу блока. Чтобы это исправить, заменим выделение памяти с помощью функции cudaMalloc() на cudaMallocPitch(). При этом модификазия программы будет не существенная. Например, в коде на рисунке 8 переменная width заменится на разницу между адресами начала строк матрицы поделенную на 4 — размер переменной типа float.

В результате такой замены количество непоследовательных обращений к памяти сократилось до нуля и время обработки одной итерации по времени составило 0.037 секунды, что в 100 раз лучше, чем для последовательной программы и в 14 раз быстрее,

чем на 8 ядрах версии программ на OpenMP. Стоит сказать, что для последовательной версии программы выравнивание начала строк давало ускорение лишь на доли процента. Оставшиеся ненулевые значения в столбце gld_incoherent связаны с тем, что помимо матриц, выравнивание начала строк которых было произведено выше, есть еще и линейные массивы, которые определяют шаг сетки по направлениям.

Таблица 1: Количество последовательных/непоследовательных операций чтения/записи после выравнивания начала строк матриц

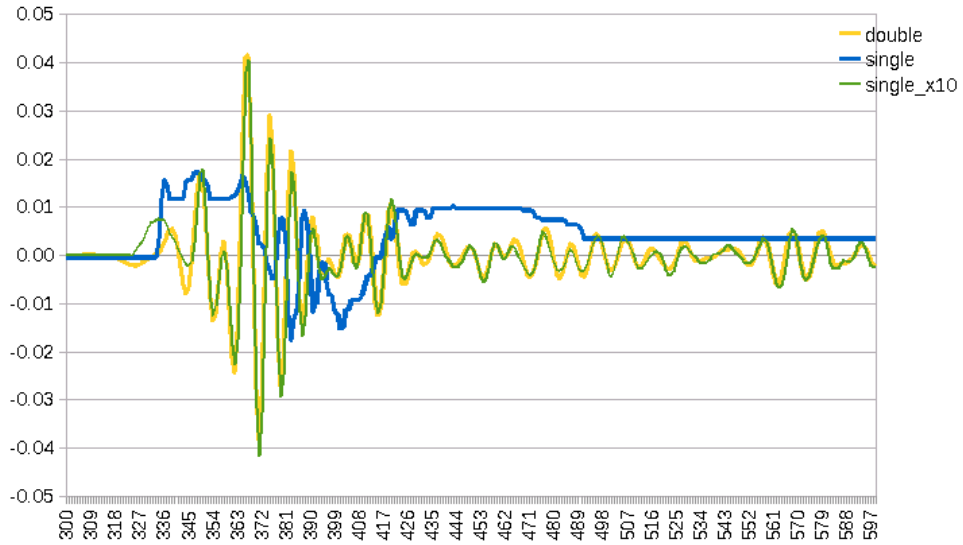
method	gld_incoherent	gld_coherent	gst_incoherent	gst_coherent
SWater_	579395	349794	0	767260
Inv2	0	174897	0	699588
SWater_r	115984	378749	0	761316
RInv	0	174900	0	233200
Inv3	0	174897	0	699588

Для программ на CPU работа с памятью происходит эффективно, если данные в памяти расположены рядом друг с другом и легко помещаются в кэш или CPU может определить шаблон доступа к памяти тем опять же заранее подгружая данные в кэш. У GPU для помещения данных в кэш можно использовать текстуры. При этом в кэш текстуры помещаются те данные, которые локализованы в двухмерном пространстве относительно данных, к которым идет обращение.

Дальнейшая оптимизация свелась к объявлению 7 двумерных текстур и 2-х одномерных, привязка текстур к областям данных в памяти GPU и небольшая модификация всех ядер. Выигрыш от такой оптимизации составил в среднем 0.03 секунды на итерацию — около 10%.

Для финального тестирования программа была выполнена на NVIDIA Tesla C1060. Полученный результат — 0.02 секунды на итерацию, что составляет итоговое ускорение около 170 раз по сравнению с исходной последовательной программой. Не смотря на то, что у Tesla C1060 в два раза больше ядер чем у GeForce 9800 GX2, ускорение в 2 раза получено не было по причине того, что тактовая частота видеокарты на 20% выше.

При тестировании на реальных данных возникли сложности: при длительном моделировании поверхность океана начинала раскачиваться и в итоге покучался некорректный результат.



В 2011 году нам удалось получить доработанную версию программы MOST. В ней были сделаны набольшие изменения, учитывающие кривизну земной прверхности, чтение и запись данных в определенном формате и выполнение всех расчетов в двойной точности.

Учитывая накопленный опыт по оптимизации данной программы и ее адаптации на разные архитектуры удалось произвести ее перенос на графические процессоры и получить ускорение в среднем в 50 раз по сравнению с исходной версией программы. Коэффициент ускорения зависит от выбранного центрального и графического процессора. При этом мы считаем, что точность вычислений остается приемлемой. Сравнение производилось как с последовательной программой на разных платформах, так и с базой данных по единичным источникам на сайте NOAA.

Original sequential program		Time [sec]/time_step
	AMD Phenom (GNU Fortran, no source code modification)	90 min
	Intel i7 (Intel Fortran compile, no vectorization, no source code modification)	28 min
	Intel i5 (GNU Fortran, no source code modification)	38 min
	Intel i3 (GNU Fortran, no source code modification)	37 min
GPU version		
	Intel i3 + GeForce GTX 550 Ti (1.8GHz)	184 sec
	AMD Phenom + GeForce GTX 550 Ti (1.8GHz)	132 sec
**	Intel i7 + NVidia Tesla C2050 (1.1GHz) PCIe x8	70sec
**	Intel i7 + NVidia Tesla C2050 (1.1GHz) PCIe x16	58sec
	Intel i5 + GeForce GTX 570 (1.56GHz) PCIe x16	44 sec

Table 2: Сравнение производительности. 4" Pacific ocean grid (pacific_4m_nocaribbean.corr)
1000 time steps, dT = 10sec, dump every 6 step,
// input data provided by Edison Giga (NOAA/PMEL)

deformation area = 200*200km

unit-source - ac_013_b

Точность моделирования составила 10^-3 см по сравнению с исходной версией программы на времени моделирования в 24 часа.