Оптимизация операции быстрого преобразования Фурье в среде OpenCL

А.А. Бурцев[1]

[1] ФГУ ФНЦ НИИСИ РАН, Москва, Россия, burtsev@niisi.msk.ru

Аннотация. Статья посвящена применению технологии OpenCL, позволяющей использовать мощные ресурсы графических процессоров для повышения быстродействия вычислительных программ. Предлагаются разнообразные варианты параллельных программ, разработанных для ускорения операции быстрого преобразования Фурье в среде OpenCL. Рассматриваемые варианты сравниваются по производительности с целью выявления наиболее оптимального для обработки комплексных векторов различной длины.

Ключевые слова: параллельное программирование, технология OpenCL, гетерогенные системы, микропроцессоры семейства КОМДИВ, быстрое преобразование Фурье

1. Введение

В настоящее время для ускорения вычислительных программ предлагаются разнообразные технологии параллельной обработки. Одни из них (такие, как, например, OpenMP [1, п.5.1]) ориентированы на создание параллельной программы, призванной исполняться на многоядерных процессорах с общей памятью. Другие (например, MPI [1, п.5.2]) позволяют создавать вычислительные программы для их исполнения на семействе компьютеров, связанных сетью.

Для ускорения вычислений в рамках одного компьютера предлагаются также специализированные процессоры SIMD класса. К ним, в частности, относятся векторный сопроцессор CPV и сопроцессор цифровой обработки сигналов CP2 семейства КОМДИВ [2], разрабатываемого в ФНЦ НИИСИ РАН.

Ещё один вид технологии, позволяющий ускорить вычислительную программу за счёт её особого распараллеливания, предполагает использование в вычислительных программах ресурсов мощных графических процессоров. Почти каждая современная видеокарта содержит целый массив однородных специализированных процессоров, которые можно использовать как слаженный ансамбль параллельно функционирующих исполнителей для ускоренного решения одной вычислительной задачи.

Первыми подобную технологию (CUDA) предложила компания NVIDIA. Аналогичную технологию, но уже пригодную для видеокарт различных семейств, предложила компания Apple. Khronos Group приняла эту технологию за основу и довела до стандарта, дав ей название **OpenCL** (**Open C**omputing **L**anguage) [3].

В новую микросхему семейства КОМДИВ включёно графическое ядро с поддержкой технологии OpenCL (версии 1.2). Для испытания возможностей этого графического ядра по технологии OpenCL в НИИСИ РАН был разработан ряд прикладных программ обработки векторов и матриц, характерных для задач линейной алгебры и цифровой обработки сигналов.

Данная статья продолжает знакомство с технологией OpenCL, начатое автором в работах [4,5]. В них рассматривались приёмы разработки в среде OpenCL эффективных программ для решения отдельных задач линейной алгебры (в том числе, операции перемножения больших матриц). А также предлагался вариант программы, позволяющей ускорить операцию быстрого преобразования Фурье (БПФ) для комплексных векторов большой длины (вида $N=2^P$).

Этот первоначальный вариант OpenCL-программы ускорения БПФ был впоследствии усовершенствован с учётом иерархического строения памяти устройств OpenCL-среды. В результате были получены новые варианты OpenCL-программы, которые существенно повысили коэффициенты ускорения операции БПФ в среде OpenCL. Применённые при создании этих вариантов приёмы оптимизации и составляют основное содержание данной статьи.

Изложение материала статьи сопровождается таблицами, в которых представлены количественные показатели ускорения операции БПФ, полученные в результате многократных прогонов разных вариантов OpenCL-программы на различных OpenCL-платформах. Эти показатели характеризуют, какие коэффициенты ускорения операции БПФ удаётся получить в среде OpenCL на обычных (настольных) компьютерах, и какое ускорение можно обеспечить в среде OpenCL, поддерживаемой микросхемой графического ускорителя семейства КОМДИВ.

2. Базовый алгоритм быстрого преобразования Фурье

Все рассматриваемые далее варианты OpenCL-программы, призванные ускорить исполнение операции БПФ в OpenCL-среде за счёт её параллельной обработки ансамблем из нескольких исполнителей, будут сравниваться по производительности (т.е. по времени выполнения) с некоторой базовой (эталонной) программой, которая рассчитана на последовательное исполнение той же операции БПФ на одном процессоре.

В качестве базового алгоритма для выполнения операции БПФ последовательной программой использовался вариант известного алгоритма Кули-Тьюки с прореживанием по времени (см. гл.12 в [6]) для векторов комплексных чисел длиной $N=2^P$ (степени двойки), который был представлен (см. п.2.2 в [5]) в виде такой Си-функции:

```
void FFT(int N,complex *X,complex *Y,
 complex *W) { complex V,T;
long int P,t,s,h,G,R,d,k,u,j,a,b;
BRVPRST(N,X,Y);//ч1.Бит-реверсная перестановка
//ч2.Основной цикл исполнения бабочек Фурье:
 P=iLog2(N); // P= log2N (так что N=2^P)
 s=1;h=2;G=1;R=N/2;d=N/2;
 for(t=1;t<=P;t++) { // на t-ом этапе:
  for(k=0,u=0;k<G;k++,u=u+d){ V=W[u];
   for(j=0,a=k;j<R;j++,a=a+h)
   { b=a+s; BF(X[a],X[b],V,T); }
  }//for k
  s=s*2;h=2*s;G=G*2;R=R/2;d=d/2;
 }//for t
}//FFT
```

Эта функция принимает исходный вектор комплексных чисел **Y** длины **N** и формирует результат преобразования Фурье как новый вектор **X**, используя для вычислений приготовленную таблицу весовых коэффициентов **W**. Она сначала получает (см. **BRVPRST)** в векторе **X** битреверсную перестановку вектора **Y**, а затем в несколько этапов (t=1,..,P, где $P=\log_2 N$) осуществляет многократные вычисления операций **БФ** («**Б**абочка **Ф**урье») над всевозможными парами элементов вектора **X**.

Заметим, что на очередном t-ом этапе (шаге цикла по t) обрабатываемые пары элементов распределяются по группам так, чтобы в каждой группе обрабатывались те пары, для которых операция БФ выполняется с одним и тем же весовым коэффициентом. На каждом шаге цикла по k для k-ой группы вычисляется индекс u для получения из таблицы W их общего весового коэффициента V=W[u], с которым далее выполняются операции БФ для всех пар этой группы, перебираемых в цикле по j.

Такое распределение пар на группы можно наглядно продемонстрировать, если представить обрабатываемый на t-м этапе вектор X как матрицу из 2R строк и G столбцов, уложенную в том же самом месте памяти по строкам. Тогда каждый столбец этой матрицы будет представлять одну группу, а соседние элементы в нём образуют R пар для исполнения операций БФ.

Сама же операция **БФ** в этой Си-функции реализована в форме макроопределения:

```
#define BF(A,B,V,T) \
 { T=B*V; B=A-T; A=A+T; }
```

Кратко охарактеризуем используемые в этом алгоритме основные переменные, которые будут далее использованы с той же ролью и в алгоритмах процедур OpenCL-ядер:

P – кол-во этапов (итераций)=$\log_2 N$, ($N=2^P$)
t – номер очередного этапа (итерации) t=1,2,..,P
На каждом t-ом этапе:
G – количество групп =$2^{(t-1)}$ = 1,2,..,N/2
k – номер очередной группы =0,1,..,G-1
R – число пар в каждой группе =$2^{(P-t)}$ = N/2,..2,1;
так что G*R=N/2 (количество всех пар)
j – номер очередной пары в группе =0,1,..,R-1
s – расстояние между элементами пары =$2^{(t-1)}$
h – расстояние между парами = 2^t =2*s
a,b – индексы элементов j-ой пары в k-ой группе: a= k+j*h; b = a+s;
d – множитель индекса для доступа в таблицу W весовых коэффициентов =$2^{(P-t)}$, (d=R)
u – индекс доступа в таблицу весовых коэффициентов W для всех пар k-ой группы = k*d

3. Первоначальный вариант OpenCL-программы для БПФ

OpenCL-программа, т.е. программа, построенная по технологии OpenCL в расчёте на её параллельное исполнение в среде OpenCL, состоит из основной программы (OpenCL-приложения на языке Си) и нескольких так называемых процедур ядра (на языке OpenCL). Основная программа запускается на основном процессоре (хосте), подготавливает OpenCL-среду и периодически запускает в ней приготовленные процедуры ядра параллельно сразу на всех её вычислительных узлах − так называемых обрабатывающих элементах (processing element).

3.1. Процедуры OpenCL-ядра для распараллеливания алгоритма БПФ

Для осуществления в среде OpenCL параллельного бит-реверсного копирования вектора приготовлена такая процедура ядра:

```
__kernel void fft_brvprst (uint L,
__global complex *X,__global complex *Y,
__global const uint *BRT) { uint k,m;
 k=get_global_id(0); m=BRT[k];
 X[m]=Y[k];// m=бит-реверсное значение k
}//fft_brvprst
```

В ней для получения бит-реверсного значения индекса **m** используется передаваемая в качестве параметра заранее приготовленная таблица **BRT**, в которой каждому целочисленному значению k=0,1,..,N-1 сопоставлено его бит-реверсное значение длины **L** (L=$\log_2$N).

Для параллельного исполнения (на очередном **t**-ом этапе) всех операций БФ над выбранными парами элементов вектора приготовлена процедура ядра:

```
__kernel void fft_btfly
(int n,__global complex *X,
 __global complex *W, int t)
{ uint i,G,R,k,j,s,h,a,b,u;
 i=get_global_id(0);// номер исполнителя (ОЭ)
 G=1<<(t-1); // кол-во групп G= 2^(t-1)
 R=n>>t; // кол-во пар в группе R= 2^(P-t);
 k=i&(G-1); // номер группы = i mod G
 j=i>>(t-1);// номер пары в группе= i div G
 s=G;h=2*s; a=k+j*h; b=a+s; u=R*k;
 BTF(X[a],X[b],W[u]);
}//fft_btfly
```

Такая процедура запускается на каждом параллельном исполнителе на **t**-ом шаге цикла, организованном в теле функции **clFFT** (см. п.3.2).

Каждый такой исполнитель, вызвав функцию **get_global_id**(0), сначала узнаёт назначенный ему уникальный номер (i=0,1,..,N/2-1). Используя параметр t как номер этапа, вычисляет по формулам: $G=2^{(t-1)}$ и $R=2^{(P-t)}$, сколько на данном этапе образуется групп G и сколько пар R в каждой группе. Затем определяет, какую пару из всех имеющихся N/2 пар ему надлежит обрабатывать: из какой она должна быть группы (k) и какая она в ней по порядку (j). Распределение пар по исполнителям осуществляется по формулам: **k= i mod G**, **j= i div G**. Тем самым гарантируется, что каждой паре обязательно будет назначен исполнитель и только один.

Далее в теле процедуры ядра вычисляются индексы (**a**,**b**) элементов выбранной пары, а также индекс **u** для взятия весового коэффициента из таблицы, заданной параметром **W**. И наконец, над назначенной парой исполняется операция «**Б**абочка **Ф**урье» (**БФ**)**,** которая в теле процедуры ядра реализована в виде вызова макроса **BTF(**X[a],X[b],W[u]**)**:

```
#define complex float2
#define BTF(XA,XB,V) \
{complex Xa,Xb,XbV; \
 Xa=XA; Xb=XB; /* XbV=Xb×V) : */  \
 XbV.x= Xb.x*V.x - Xb.y*V.y; \
 XbV.y= Xb.x*V.y + Xb.y*V.x; \
 XA= Xa + XbV; XB= Xa - XbV; }
```

Эти процедуры ядра вместе с необходимыми макроопределениями сосредоточены в файле “FFT.cl”, строки которого анализируются при компоновке программного кода ядра в основной программе.

3.2. Основная программа для запуска исполнения БПФ в среде OpenCL

В основной программе сначала осуществляются все необходимые действия по подготовке OpenCL-среды: инициализируется OpenCL-платформа, дескрипторы OpenCL-устройств, подготавливается OpenCL-контекст, в котором создаётся дескриптор очереди команд.

Затем приготавливаются все объекты для представления в памяти OpenCL-данных, подлежащих обработке, и особый объект **prgrm**, содержащий в откомпилированной форме код всех процедур ядра. И для каждой процедуры ядра создаётся дескриптор типа cl_kernel:

```
kn1=clCreateKernel (prgrm, "fft_brvprst",NULL);
kn2=clCreateKernel (prgrm, "fft_btfly", NULL);
```

Как обеспечить все эти действия, было подробно описано в [4] (в п.2.3-2.5) и в [5] (в п.3.2).

Для выполнения в среде OpenCL операции БПФ оформлена функция **clFFT**:

```
void clFFT(int N,complex *X,complex *Y)
```

в теле которой предусмотрена последовательность действий из следующих 6-ти частей.

1. Загрузка вектора Y в буфер объекта **objY**:

```
cl_uint szC= sizeof(complex);
clEnqueueWriteBuffer(cmndQ,
  objY,CL_TRUE,0,szC*N,Y,0,0,0);
```

2. Назначение 4-х фактических параметров для процедуры ядра fft_brvprst:

```
cl_uint szI,szM; cl_uint P=iLog2(N);
szI=sizeof(cl_uint); szM=sizeof(cl_mem);
clSetKernelArg(kn1,0,szI,&P);
clSetKernelArg(kn1,1,szM,&objX);
clSetKernelArg(kn1,2,szM,&objY);
clSetKernelArg(kn1,3,szM,&objBRT);
```

3. И запуск её в OpenCL-среде на множестве из N исполнителей для параллельного бит-реверсного копирования вектора из буфера объекта **objY** в буфер объекта **objX** с ожиданием её завершения всеми исполнителями:

```
size_t gWS[1]={N}; cl_event ev1;
clEnqueueNDRangeKernel(cmndQ,
 kn1,1,NULL,gWS,NULL,0,NULL,&ev1);
clFinish(cmndQ);
```

4. Подготовка к многократному запуску процедуры ядра fft_btfly с назначением для неё 3-х первых фактических параметров:

```
clSetKernelArg(kn2,0,szI,&N);
clSetKernelArg(kn2,1,szM,&objX);
clSetKernelArg(kn2,2,szM,&objW);
```

5. Организация основного цикла (по t), на каждом шаге которого процедура ядра fft_btfly запускается на множестве из N/2 исполнителей для параллельного выполнения всех операций БФ t-го этапа (t=1,..,P) с передачей ей значения номера этапа t в качестве 4-го параметра и ожиданием её завершения всеми исполнителями:

```
cl_uint t; cl_event ev2; gWS[0]=N/2;
```

```
for (t=1; t<=P; t++) {
 clSetKernelArg(kn2,3,szI,&t);
 clEnqueueNDRangeKernel(cmndQ,
 kn2,1,NULL,gWS,NULL,0,NULL,&ev2);
 clWaitForEvents(1,&ev2);
}//for t
```

6. Выгрузка в **X** результирующего вектора, полученного в буфере объекта **objX**:

```
clEnqueueReadBuffer(cmndQ,
 objX,CL_TRUE,0,szC*N,X,0,0,0);
```

Этот первоначально созданный вариант OpenCL-программы обозначим как вариант **A**. И оценим его производительность.

3.3. Результаты ускорения операции БПФ в среде OpenCL варианта A

Чтобы оценить, насколько удалось ускорить БПФ с помощью OpenCL-программы, основная программа OpenCL-приложения сначала запускала функцию FFT для обычного (последовательного) исполнения операции БПФ, а затем подготавливала OpenCL-среду и запускала в ней функцию clFFT для выполнения той же операции в среде параллельных исполнителей.

Первоначальный вариант (**A**) программы был разработан (на языке Си) как консольное приложение в среде MS Visual Studio 2017. Он был скомпонован и запускался на различных OpenCL-платформах.

1. Под ОС Windows-10 для платформы с аппаратной конфигурацией, содержащей основной универсальный процессор CPU Intel-i59400 с частотой 2.9 Ггц и встроенный графический процессор GPU UHD 630 с частотой 350 МГц. (далее будем называть её «платформа **Intel**»).

2. Под ОС Windows-7 с аппаратной конфигурацией, содержащей универсальный процессор Intel i3-2100 с частотой 3.1 Ггц и видеокарту NVidia GeForce 1050ti с частотой 1392 Мгц. (далее будем называть её «платформа **NVidia**»).

Скомпонованная программа многократно прогонялась на этих платформах для векторов комплексных чисел одинарной точности различной длины $N=2^P$ (P=8,9,..,24). Результаты этих прогонов были детально изложены в [5]. Здесь же кратко представим результаты этого первоначального варианта A в виде единой сводной таблицы, поместив в неё лишь итоговые коэффициенты так называемого общего и «чистого» ускорения операции БПФ на векторах различной длины (см. таблицу 1). Смысл и формулы расчета этих коэффициентов поясняются внизу приводимой таблицы.

Полученные результаты практически подтверждают, что применение технологии OpenCL действительно позволяет существенно ускорить операцию БПФ для векторов комплексных чисел большой длины вида $N=2^P$ (P=8,9,..,23,24).

Таблица 1. Ускорение операции БПФ OpenCL-программой варианта **A** на платформах Intel и NVidia

P	$N=2^P$	Intel		NVidia	
		K1	K2	K1	K2
8	256	0.09	1.94	0.18	4.59
9	512	0.18	3.92	0.29	6.86
10	1024	0.38	7.96	0.67	15.01
11	2048	0.75	14.40	**1.75**	31.88
12	4096	**1.49**	25.53	3.33	61.85
13	8192	3.02	42.69	5.67	129.35
14	16384	5.68	58.13	14.80	202.75
15	32768	9.83	**76.37**	31.20	299.08
16	65536	18.66	74.58	46.00	**420.17**
17	131072	29.40	75.56	50.00	293.10
18	262144	34.86	67.20	80.25	252.25
19	524288	35.51	51.80	139.33	303.69
20	1048576	39.92	50.93	161.45	303.52
21	2097152	40.89	47.19	160.58	309.70
22	4194304	36.78	40.27	179.70	310.10
23	8388608	35.55	37.87	186.67	312.86
24	16777216	33.55	35.23	**192.70**	**311.90**

$\mathbf{T_{CPU}}$ – время исполнения на CPU
$\mathbf{T_{CL}}$ – полное время исполнения в OpenCL-среде
$\mathbf{T_{Я}}$ – время исполнения OpenCL-процедур ядра
K1 –коэффициент общего ускорения = $\mathbf{T_{CPU}/T_{CL}}$
K2 –коэффициент «чистого» ускорения =$\mathbf{T_{CPU}/T_{Я}}$

3.4. Недостатки OpenCL-программы первоначального варианта

Рассмотренный первоначальный вариант (**A**) OpenCL-программы имеет ряд недостатков, избавившись от которых, можно рассчитывать получить более совершенные версии программы для ускорения операции БПФ. Эти недостатки обусловлены тем, что представленный вариант OpenCL-программы не использует в полной мере все возможности, предоставляемые средой OpenCL.

Запускаемые этой программой процедуры ядра не извлекают никакой выгоды из неоднородного строения памяти OpenCL-устройства. Слабо используют самую быстродействующую внутреннюю память самих обрабатывающих элементов (ОЭ) и совсем не используют локальную память рабочей группы, в которую можно объединять несколько ОЭ для совместной работы. Из-за этого каждый параллельный исполнитель вынужден всякий раз обращаться за данными в общую глобальную память, что, конечно же, замедляет их совместную работу.

К тому же для исполнения второй части БПФ организуется цикл, в котором на каждом шаге процедура ядра fft_btfly запускается заново на всех обрабатывающих элементах. Из-за чего увеличиваются накладные расходы, затрачиваемые на запуск параллельных исполнителей и синхронизацию их совместной работы.

Далее рассмотрим другие варианты OpenCL-программы, направленные на устранение отмеченных недостатков.

4. Вариант OpenCL-программы с одной рабочей группой

Будем запускать параллельные исполнители, выполняющие процедуры ядра, объединяя их все в одну рабочую группу (Working Group). Воспользовавшись средствами синхронизации, предоставляемыми рабочей группой, составим такую процедуру ядра, которая будет запускаться однократно, и сама будет организовывать весь цикл по этапам (цикл по t) для выполнения 2-ой части операции БПФ:

```
__kernel void fft_btflyLoopB
(int n,__global complex *X,
 __global complex *W, int P) {
uint i,G,R,k,j,s,h,a,b,u;
i=get_global_id(0);// номер исполнителя (ОЭ)
for(t=1;t<=P;t++){
 G=1<<(t-1); // кол-во групп G= 2^(t-1)
 R=n>>t; // кол-во пар в группе R= 2^(P-t);
 k=i&(G-1); // номер группы = i mod G
 j=i>>(t-1);// номер пары в группе= i div G
 s=G; h=2*s; a=k+j*h; b=a+s; u=R*k;
 BTF(X[a],X[b],W[u]);
 barrier(CLK_LOCAL_MEM_FENCE);
}//for t
}//fft_btflyLoopB
```

Каждый исполнитель такой процедуры в конце очередного шага цикла будет вызывать функцию **barrier**, чтобы синхронизировать свою дальнейшую работу с другими исполнителями рабочей группы и дождаться момента, когда они все вместе закончат этот шаг цикла.

Добавим в OpenCL-программу действия по созданию дескриптора новой процедуры ядра:

```
kn3=clCreateKernel(prgrm,"fft_btflyLoopB",NULL);
```

Изменим 4-й пункт действий функции clFFT:

```
cl_event ev3;
size_t lWS[1]={N/2}; gWS[0]=N/2;
clSetKernelArg(kn3,0,szI,&N);
clSetKernelArg(kn3,1,szM,&objX);
clSetKernelArg(kn3,2,szM,&objW);
clSetKernelArg(kn3,3,szI,&P);
```

И 5-й пункт списка действий функции clFFT:

```
clEnqueueNDRangeKernel(cmndQ,
 kn3,1,NULL,gWS,lWS,0,NULL,&ev3);
clWaitForEvents(1,&ev3);
```

Полученный вариант OpenCL-программы обозначим как вариант **B**. Результаты прогонов этого варианта программы на платформах Intel и NVidia представлены в таблице 2.

Таблица 2. Ускорение операции БПФ OpenCL-программой варианта **B** на платформах Intel и NVidia

P	$N=2^P$	Intel		NVidia	
		K1	K2	K1	K2
8	256	0.39	5.71	0.40	10.41
9	512	0.85	10.86	1.0	23.39
10	1024	---	---	2.00	45.76
11	2048	---	---	**4.67**	79.61

Сравнивая полученные «чистые» коэффициенты ускорения (K2) с теми, что обеспечивались вариантом A (в таблице 1), можно заметить, что программа варианта B значительно их улучшает: почти в 2 раза на платформе Intel и примерно в 3 раза на платформе NVidia.

Однако, эта OpenCL-программа варианта B обеспечивает такое улучшение лишь для векторов небольшого размера (длины $N \le 512$ для платформы Intel и $N \le 2048$ для платформы NVidia). Вектора же б**о**льшего размера программой варианта B обрабатываться, увы, не могут. Почему, в чём причина?

Дело в том, что для исполнения процедуры ядра fft_btflyLoopB параллельно запускается N/2 обрабатывающих элементов, которые все вместе должны быть объединены в одну рабочую группу. А количество элементов, которое можно собрать в одну группу, ограничено: оно не может превышать значения 256 для платформы Intel, и 1024 – для платформы NVidia.

Как же быть? Можно ли путём организации рабочих групп всё же повысить коэффициент ускорения и для векторов больших размеров?

5. Вариант OpenCL-программы со многими рабочими группами

Допустим, заранее известно, какое максимальное число (MW) обрабатывающих элементов можно объединить в одну рабочую группу. Подберём подходящее значение M для размера группы так, чтобы оно было степенью двойки ($M=2^q$) и не превышало разрешённый максимум: MW=256 (2^8) для платформы Intel и MW=1024 (2^{10}) для платформы NVidia.

Собранные в одну рабочую группу M рабочих элементов (working items) способны совместно исполнить первые несколько (PB) этапов 2-ой части БПФ над всеми парами элементов из выделенного группе непрерывного участка вектора длиной $NB=2 \cdot M$ ($PB=\log_2 NB$). Разбив весь вектор на участки длиной NB, можно каждый участок обрабатывать отдельной рабочей группой, и тем самым, хотя бы на этой первой стадии, охватывающей первые PB этапов (для t=1,..,PB), воспользоваться преимуществами рабочих групп для ускорения вычислительной обработки. А затем на второй стадии оставшиеся этапы (t=PB+1,..,P) можно обрабатывать прежним способом, как в варианте A.

Для выполнения первой стадии составим новую процедуру ядра (fft_btflyLoopC), которую будем запускать на всех обрабатывающих элементах всех рабочих групп. Схема назначения пар исполнителям и алгоритм вычисления параметров пары в этой процедуре поясняются на рисунке 1.

```
__kernel void fft_btflyLoopC
(int n,__global complex *X,
 __global complex *W, int PB) {
uint G,R,k,j,s,h,a,b,u;
uint g,i,WS,kl,jl,kG,jG;
i=get_local_id(0);//локальный номер исполнителя
WS=get_local_size(0);// размер рабочей группы
g=get_group_id (0);// номер рабочей группы
kG=0; jG= g*WS; /* jG=g*WS/G , но G=1*/
for(t=1;t<=PB;t++){
 G=1<<(t-1); R=n>>t; s=G; h=2*s;
 kl=i&(G-1);jl=i>>(t-1);//kl= i%G; jl=i/G
 k=kG+kl;j=jG+jl; u=R*k;
 a=k+j*h; b=a+s; jG=jG>>1;
 BTF(X[a],X[b],W[u]);
 barrier(CLK_LOCAL_MEM_FENCE);
}//for t
}//fft_btflyLoopC
```

В этой процедуре требуется обеспечить, чтобы на каждом этапе исполнители одной и той же рабочей группы выбирали свою пару элементов именно того участка вектора, который поручено обрабатывать данной рабочей группе. И параметры **k** и **j**, задающие пару элементов, для которой данному исполнителю (с локальным номером **i** в рабочей группе **g**) надлежит выполнить операцию БФ на **t**-ом этапе, вычисляются немного сложнее. Сначала определяются их локальные представители в пределах рабочей группы: **kl**= **i mod G**, **jl**= **i div G**, а потом уже сами **k**=**kG**+**kl** и **j**=**jG**+**jl**. При этом **kG** и **jG** задают базовые номер группы элементов и номер пары в этой группе, с которых начинается перечисление всех пар элементов выделенного этой рабочей группе участка вектора.

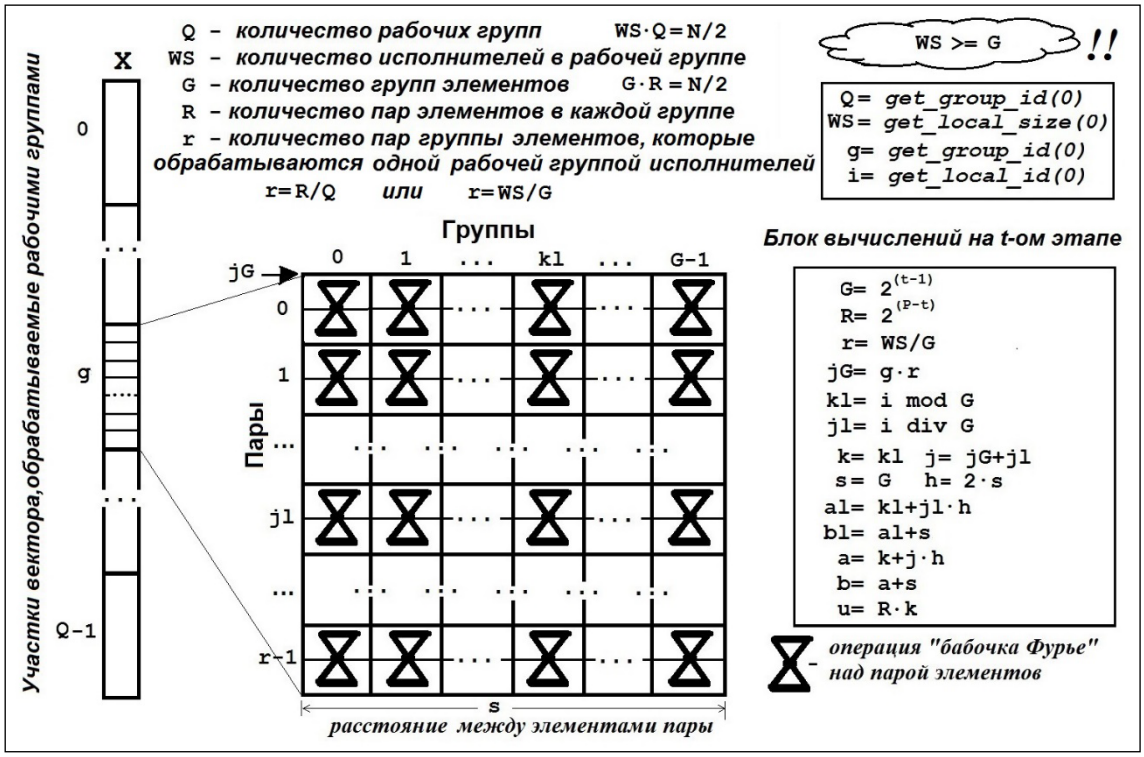

Рис. 1. Схема назначения пар в процедуре ядра 1-й стадии варианта C

Внесём необходимые изменения в OpenCL-программу. Подкорректируем имя новой процедуры ядра при создании для неё дескриптора:

```
kn3=clCreateKernel(prgrm,"fft_btflyLoopC",NULL);
```

Добавим в 4-й пункт функции clFFT назначение параметров для новой процедуры ядра:

```
cl_uint PB=Log2(NB);
clSetKernelArg(kn3,0,szI,&N);
clSetKernelArg(kn3,1,szM,&objX);
clSetKernelArg(kn3,2,szM,&objW);
clSetKernelArg(kn3,3,szI,&PB);
```

И составим 5-й пункт списка действий функции clFFT, выделив в нём две стадии:

```
cl_uint t; cl_event ev2,ev3;
size_t lWS[1]={NB/2}; gWS[0]=N/2;
{ 1-ая стадия } // t=1,..,PB; PB= log2(NB);
clEnqueueNDRangeKernel(cmndQ,
 kn3,1,NULL,gWS,lWS,0,NULL,&ev3);
clWaitForEvents(1,&ev3);
{ 2-ая стадия } // t=PB+1,..,P; P= log(N);
for (t=PB+1; t<=P; t++){
 clSetKernelArg(kn2,3,szI,&t);
 clEnqueueNDRangeKernel(cmndQ,
 kn2,1,NULL,gWS,NULL,0,NULL,&ev2);
 clWaitForEvents(1,&ev2); }//for t
```

Полученный вариант OpenCL-программы обозначим как вариант **C**. Результаты прогонов этого варианта программы на платформах Intel и NVidia представлены в таблице 3.

Сравнивая их «чистые» коэффициенты ускорения (K2) с вариантом A, можно заметить, что они по-прежнему (как и в варианте B) значительно увеличились (в 2-3 раза) лишь для векторов небольшой длины. С ростом же длины вектора (N) процент повышения коэффициента K2 становится всё меньше. И если для векторов длины N=8192 он ещё повышается на 54% на платформе Intel и на 87% на платформе NVidia, то при N=8388608 (2^{23}) коэффициент K2 варианта C превосходит K2 варианта A всего лишь на 10% (Intel) и 13% (NVidia) соответственно.

Таблица 3. Ускорение операции БПФ OpenCL-программой варианта **C** на платформах Intel и NVidia

P	**N=2^P**	**Intel**		**NVidia**	
		K1	**K2**	**K1**	**K2**
8	256	0.39	5.70	0.40	10.40
9	512	0.87	10.96	1.25	23.50
10	1024	1.31	18.79	3.00	46.21
11	2048	2.17	30.26	3.50	79.69
12	4096	3.77	46.98	7.75	138.21
13	8192	6.56	59.97	11.33	240.73
14	16384	11.42	79.26	24.33	319.15
15	32768	18.57	**95.78**	52.33	421.79
16	65536	29.36	85.77	70.00	515.48
17	131072	41.92	83.90	150.50	341.24
18	262144	**46.25**	73.78	160.00	306.92
19	524288	35.64	51.55	159.80	342.92
20	1048576	39.99	50.13	177.90	353.94
21	2097152	37.38	43.02	192.40	355.49
22	4194304	37.64	40.97	205.11	**372.95**
23	8388608	35.34	37.86	208.05	352.56
24	16777216	34.18	36.04	**224.57**	367.75

Выигрыш использования программы варианта C на векторах большой длины оказывается, увы, не таким значительным, как для векторов малых размеров. Чем это можно объяснить? Объяснение простое: с ростом N увеличивается доля той обработки, которая выполняется на 2-й стадии, а доля 1-ой стадии, для ускорения которой и были применены рабочие группы, соответственно уменьшается, поэтому и общий выигрыш в повышении коэффициента K2 становится всё меньше.

Возникает вопрос: можно ли применить рабочие группы и на 2-й стадии тоже? Ответ: можно, но потребуется усложнить запускаемую процедуру ядра. И такой усложнённый вариант программы мы рассмотрим далее (в п.7). Но прежде попробуем оптимизировать полученную OpenCL-программу варианта C, применяя ещё одну полезную возможность, которую обеспечивает рабочая группа. Речь пойдёт об использовании локальной памяти.

6. Использование локальной памяти в рабочих группах

Рабочим элементам, т.е. обрабатывающим элементам, объединённым в одну и ту же рабочую группу, предоставляется общая для них всех локальная память. Предполагается, что доступ к ней осуществляется быстрее, чем к глобальной памяти OpenCL-устройства. Поэтому для повышения общей производительности следует стараться сохранять в этой локальной памяти ту часть элементов вектора, которая наиболее активно обрабатывается всеми исполнителями рабочей группы.

Учитывая это, модифицируем процедуру ядра, запускаемой на первой стадии, следующим образом. Пусть на первом этапе (при **t**=1) каждый исполнитель группы, вытащив из глобальной памяти пару элементов (с индексами **a** и **b**) и выполнив с ними операцию БФ, помещает результаты в локальную память (с индексами **al** и **bl**). На последующих этапах (при **t**=2,..,PB-1) все исполнители совершают операции БФ со своими парами, взятыми из локальной памяти. А на последнем этапе (при **t**=PB), взяв элементы из локальной памяти, каждый исполнитель помещает результат операции БФ обратно в глобальную память.

Выполнив такую модификацию, получим следующую процедуру ядра (fft_btflyLoopCL):

```
kernel void fft_btflyLoopCL
(int n,__global complex *X,
 __global complex *W, int PB
 __local complex *XL ){
uint G,R,WS,k,j,s,h,a,b,u;
uint g,i,kG,jG, kl,jl,al,bl;
i=get_local_id(0);//локальный номер исполнителя
WS=get_local_size(0); // размер раб.группы
g=get_group_id (0); // номер раб.группы
kG=0; jG=g*WS; /* jG=g*WS/G , но G=1*/
get_GRkjuabL(1); a=k+j*h; b=a+s;
if(PB==1){BTF(X[a],X[b],W[u]);}else{
BTFLY(X[a],X[b],W[u],XL[al],XL[bl]);
for(t=2;t<=PB-1;t++){
 barrier(CLK_LOCAL_MEM_FENCE);
 get_GRkjuabL(t);
 BTF(XL[al],XL[bl],W[u]);
}//for t
 barrier(CLK_LOCAL_MEM_FENCE);
 get_GRkjuabL(PB); a=k+j*h;b=a+s;
 BTFLY(XL[al],XL[bl],W[u],X[a],X[b]);
}//if PB==1
}//fft_btflyLoopCL
```

В этой процедуре на каждом этапе для определения параметров обрабатываемой пары кроме переменных **G**,**R**, **k**,**j**, **u**, **a**,**b**, **kl**, **jl**, необходимо ещё и вычислять индексы **al**,**bl**, задающие места расположения элементов пары в локальной памяти (см. рис. 1). Чтобы не записывать такой блок вычислений несколько раз, предложено

макроопределение **get_GRkjuabL**, которое вызывается в этой процедуре 3 раза:

```
#define get_GRkjuabL(t) { \
 G=1<<(t-1); R=n>>t; s=G; h=2*s; \
 kl=i&(G-1); jl=i>>(t-1); \
 k=kG+kl; j=jG+jl; u=R*k; \
 al=kl+jl*h; bl=al+s; jG=jG>>1; }
```

Отметим также, что на первом и последнем этапе для исполнения операции БФ применяется другое макроопределение **BTFLY**, которое позволяет поместить результат операции в другое место памяти:

```
#define complex float2
#define BTFLY(XA,XB,V,nXA,nXB) \
{complex Xa,Xb,XbV; \
 Xa=XA; Xb=XB; /* XbV=Xb×V) : */  \
 XbV.x= Xb.x*V.x - Xb.y*V.y; \
 XbV.y= Xb.x*V.y + Xb.y*V.x; \
 nXA= Xa + XbV; nXB= Xa - XbV; }
```

И ещё заметим, что в этой процедуре отдельно обрабатывается особый случай, когда процедура ядра должна выполнить лишь один этап (PB=1). В этом случае результат операции БФ должен сразу помещаться в глобальную память, и локальная память тогда не используется.

Изменим немного и OpenCL-программу. Поменяем имя процедуры ядра:

```
kn3=clCreateKernel(prgrm,"fft_btflyLoopCL",0);
```

И добавим назначение 5-го параметра (**XL**) процедуры ядра, задав в качестве его характеристики требуемый размер (szC·NB) для вектора комплексных чисел, не уточняя, где именно он будет располагаться в локальной памяти:

```
clSetKernelArg(kn3,4,szC*NB,NULL);
```

Полученный вариант OpenCL-программы обозначим как вариант $\mathbf{C^L}$. Результаты прогонов этого варианта программы на платформах Intel и NVidia представим в таблице 4.

Таблица 4. Ускорение операции БПФ OpenCL-программой варианта $\mathbf{C^L}$ для платформ Intel и NVidia

P	$N=2^P$	Intel		NVidia	
		K1	K2	K1	K2
8	256	0.39	4.90	0.50	15.66
9	512	0.87	10.05	0.83	26.00
10	1024	1.16	17.90	2.00	56.31
11	2048	2.18	29.76	4.67	99.88
12	4096	3.74	45.93	7.50	158.97
13	8192	6.22	64.29	17.00	286.46
14	16384	11.38	78.70	37.00	421.76
15	32768	17.83	95.11	52.00	541.96
16	65536	30.43	85.08	69.00	666.94
17	131072	40.91	83.57	75.00	392.85
18	262144	45.39	72.70	107.00	346.71
19	524288	41.71	54.94	167.20	399.06
20	1048576	45.78	54.40	197.33	388.94
21	2097152	47.02	52.95	192.70	389.61
22	4194304	40.89	44.40	206.65	383.04
23	8388608	39.24	41.37	213.33	380.57
24	16777216	36.23	37.78	221.83	367.68

Чтобы наглядно ощутить, насколько использование локальной памяти рабочих групп позволяет повысить коэффициенты ускорения, приведём сравнительную таблицу (см. таблицу 5), в которой для OpenCL-программ вариантов C и C^L, опробованных на платформе NVidia, представлены «чистые» коэффициенты ускорения K2 вместе с показателями, характеризующими их рост по отношению к коэффициентам K2 варианта A.

Таблица 5. Сравнение коэффициентов ускорения операции БПФ вариантов **C** и $\mathbf{C^L}$ платформы NVidia

P	$N=2^P$	вариант **C**		вариант $\mathbf{C^L}$	
		K2	**K2^**	**K2**	**K2^**
8	256	10.40	3.41	15.66	3.41
9	512	23.50	3.42	26.00	**3.79**
10	1024	46.21	3.09	56.31	3.75
11	2048	79.69	2.51	99.88	3.13
12	4096	138.21	2.17	158.97	2.57
13	8192	240.73	**1.87**	286.46	**2.21**
14	16384	319.15	1.60	421.76	2.08
15	32768	421.79	1.40	541.96	1.81
16	65536	515.48	1.21	666.94	1.59
17	131072	341.24	1.17	392.85	1.34
18	262144	306.92	1.22	346.71	1.37
19	524288	342.92	1.18	399.06	1.31
20	1048576	353.94	1.17	388.94	1.28
21	2097152	355.49	1.15	389.61	1.26
22	4194304	372.95	1.14	383.04	1.24
23	8388608	352.56	**1.13**	380.57	**1.22**
24	16777216	**367.75**	1.13	367.68	1.18

K2 – коэффициент ускорения = $\mathbf{T_{CPU}}$ / **Tя**
K2^ – коэффициент роста = **K2** / **K2** варианта **A**

Из этой таблицы видно, что использование локальной памяти заметно улучшает коэффициент ускорения K2 при обработке векторов любой длины. Так, например, для векторов длины N=8388608 (2^{23}) этот коэффициент подрастает уже на 22% (вместо 13% варианта C), а для векторов длины N=8192 – на 121% (вместо 87%).

Поэтому далее во всех вариантах будем активно использовать локальную память рабочих групп везде, где это возможно. Например, можно аналогичным образом модифицировать процедуру ядра варианта B, чтобы она тоже работала с локальной памятью (полученный вариант OpenCL-программы назовём вариантом $\mathbf{B^L}$).

7. Двойное применение рабочих групп с локальной памятью

Итак, применение рабочих групп с локальной памятью позволило заметно ускорить первую стадию операции БПФ (на этапах t=1,..,PB). Попробуем усилить полученный эффект и применить этот же приём оптимизации и на второй стадии (этапах t= PB+1,..,P) тоже.

На 1-ой стадии БПФ каждой рабочей группе, состоящей из M исполнителей, выделялся для совместной обработки непрерывный участок вектора из NB=2·M элементов, образующих M пар на каждом этапе. На 2-ой стадии каждой рабочей группе тоже предстоит вместе поэтапно обрабатывать одну и ту же цепочку из нескольких (N/NB) элементов. Но таких элементов, которые уже не стоят в заданном векторе непрерывно друг за другом, а следуют в нём с некоторым постоянным шагом (**s**=$2^{\mathbf{PB}}$).

На каждом этапе требуется корректно распределять пары этих элементов между исполнителями группы. А для этого в процедуре ядра необходимо задать формулы, по которым на этапе **t** исполнитель с локальным номером **i** мог бы правильно вычислить параметры (**k**,**j**,**a**,**b**,**u**) именно той пары элементов, которая должна назначаться ему для исполнения операции БФ. Для цепочки элементов, разбросанных по вектору, выразить такие формулы становится сложнее, чем для сплошного блока элементов, как это было в процедуре ядра для 1-ой стадии.

Представим вектор как матрицу, размещённую в том же месте памяти по строкам размером по **s** элементов. Тогда цепочки элементов, которую должны обрабатывать исполнители одной и той же рабочей группы, окажутся столбцами этой матрицы (см. рис. 2).

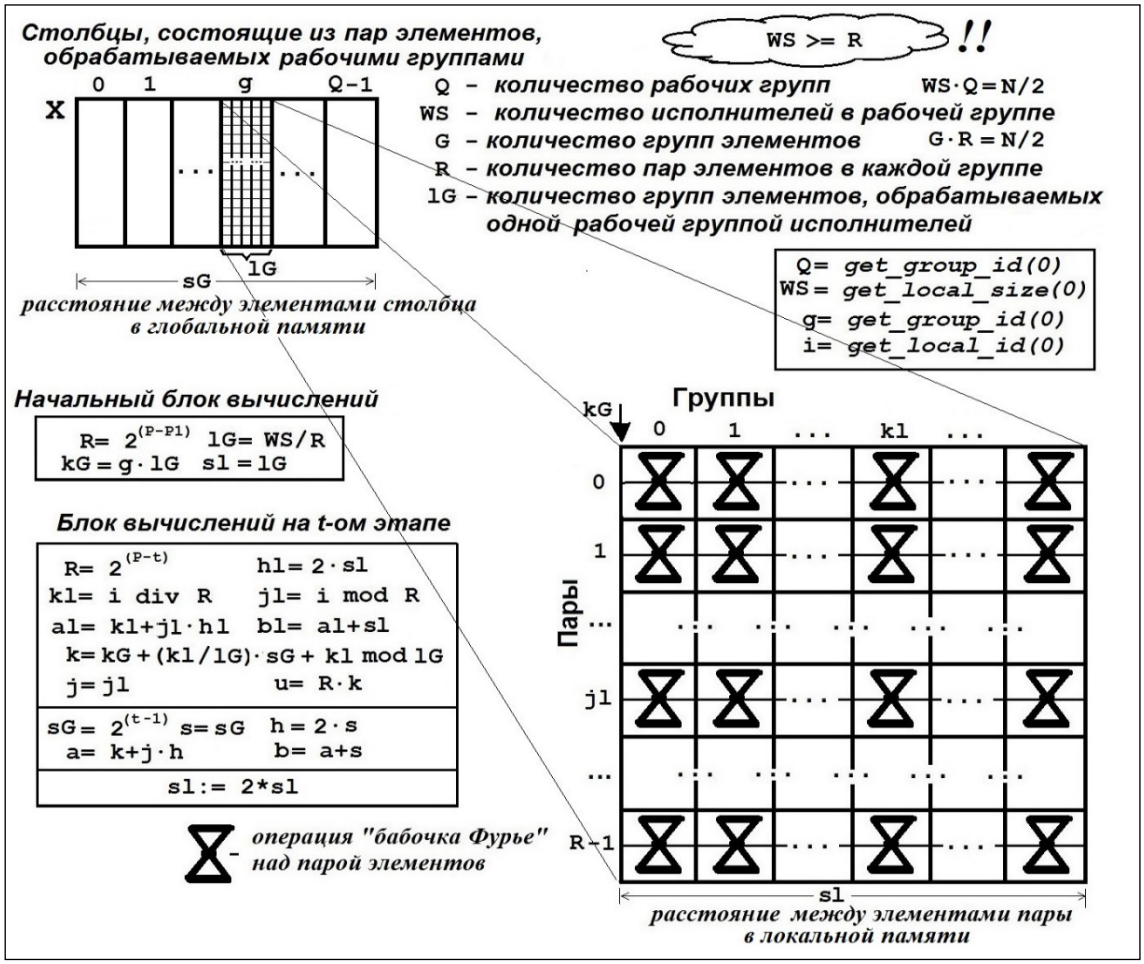

Рис. 2. Схема назначения пар в процедуре ядра 2-й стадии варианта D

Такое представление позволяет пояснить алгоритм вычисления параметров назначаемой пары в процедуре ядра, вызываемой на 2-ой стадии:

```
kernel void fft_btflyLoopTL
(int n,__global complex *X,
 __global complex *W, int P1,
 int P2, __local complex *XL ){
uint lG,R,k,j,sl,h,a,b,u;
uint g,i,WS,kl,jl,kG,sG;
i=get_local_id(0);//локальный номер исполнителя
WS=get_local_size(0);// размер раб.группы
g=get_group_id (0);// номер раб.группы
R=n>>P1; sG=1<<(P1-1);
lG=WS/R; //предполагается, что WS >= R !! */
kG=g*lG;//lG=к-во групп эл-тов в рабочей группе
sl=lG; get_RkjuabL(P1); sl=sl<<1;
a=k+j*2*sG; b=a+sG;
if(P1==P2){BTF(X[a],X[b],W[u]);}else{
BTFLY(X[a],X[b],W[u],XL[al],XL[bl]);
for(t=P1+1;t<=P2-1;t++){
 barrier(CLK_LOCAL_MEM_FENCE);
 get_RkjuabL(t); sl=sl<<1;
 BTF(XL[al],XL[bl],W[u]);
}//for t
 barrier(CLK_LOCAL_MEM_FENCE);
 get_RkjuabL(P2); sG=1<<(P2-1);
 a=k+j*2*sG; b=a+sG;
BTFLY(XL[al],XL[bl],W[u],X[a],X[b]);
}//if P1==P2
}//fft_btflyLoopTL
```

Для вычисления параметров обрабатываемой на очередном этапе пары в этой процедуре ядра используется другое макроопределение:

```
#define get_RkjuabL(t) { \
 R=n>>t; jl= i % R; kl= i / R; \
 k=kG+(kl/lG)*sG+kl%lG; j=jl; \
 al=kl+2*jl*sl; bl=al+sl; u=R*k; }
```

Дополним OpenCL-программу, чтобы получить дескриптор для новой процедуры ядра:

```
kn4=clCreateKernel(prgrm,"fft_btflyLoopTL",NULL);
```

Назначим ей параметры при выполнении 4-го пункта списка действий функции clFFT:

```
cl_uint PB1=PB+1;
clSetKernelArg(kn4,0,szI,&N);
clSetKernelArg(kn4,1,szM,&objX);
clSetKernelArg(kn4,2,szM,&objW);
clSetKernelArg(kn4,3,szI,&PB1);
clSetKernelArg(kn4,4,szI,&P);
clSetKernelArg(kn4,5,szC*(N/NB),NULL);
```

И чтобы запускать такую процедуру ядра на 2-ой стадии БПФ, поставим в 5-й пункт списка действий функции clFFT (вместо цикла по t) вот такой фрагмент:

```
{ 2-ая стадия }// t=PB+1,..,P; P= log(N);
lWS[0]= (N/NB)/2;
clEnqueueNDRangeKernel(cmndQ,
 kn4,1,NULL,gWS,lWS,0,NULL,&ev4);
clWaitForEvents(1,&ev4);
```

После произведённых преобразований получим новый вариант OpenCL-программы, который будем называть далее вариантом **D**. Поскольку он отличается от предыдущего варианта C^L иным исполнением 2-ой стадии БПФ, то и ощутить разницу в производительности этих вариантов можно будет лишь на векторах такой длины, при обработке которых эта 2-я стадия исполняется. Приведём сравнительную таблицу коэффициентов ускорения этих вариантов для платформы NVidia, где эта разница может проявиться при длине N≥4096 (см. таблицу 6).

Таблица 6. Сравнение коэффициентов ускорения операции БПФ вариантов $\mathbf{C^L}$ и **D** платформы NVidia

P	$N=2^P$	вариант C^L		вариант **D**	
		K2	**K2^**	**K2**	**K2^**
12	4096	158.97	2.57	102.97	1.66
13	8192	286.46	2.21	187.78	1.45
14	16384	421.76	2.08	299.09	1.48
15	32768	541.96	1.81	445.64	1.49
16	65536	666.94	1.59	608.41	1.45
17	131072	392.85	1.34	**545.52**	**1.86**
18	262144	346.71	1.37	495.14	1.96
19	524288	399.06	1.31	576.73	1.90
20	1048576	388.94	1.28	566.46	1.87
21	2097152	389.61	1.26	562.78	1.82
22	4194304	383.04	1.24	521.69	1.68

Как видно из этой таблицы, вариант D действительно улучшает коэффициент «чистого» ускорения (K2) операции БПФ, но лишь для векторов длиной $N\geq 2^{17}$(=131072).

Получается, что произведённая модификация 2-ой стадии БПФ путём замены итерационного запуска процедуры ядра fft_btfly на одиночный запуск процедуры ядра fft_btflyoopTL с применением рабочих групп и локальной памяти оказывается оправданной в том случае, если эта 2-ая стадия состоит как минимум из 6-ти этапов. А при меньшем числе этапов усложнённые вычисления параметров назначаемых пар, исполняемые процедурой fft_btflyoopTL, проигрывают по быстродействию циклическим вызовам процедуры fft_btfly, в которой параметры предназначенной исполнителю пары определяются значительно проще.

Заметим также, что вариант D не может применяться для векторов с длиной $N>2^{22}$ (на платформе NVidia) и $N>2^{18}$ (на платформе Intel). Возникает вопрос, почему? Всё дело в том, что в этом варианте операция БПФ ограничивается проведением 2-х стадий, каждая из которых может охватывать не более PW этапов. Причём, величина PW зависит от максимально допустимого количества (MW) исполнителей (рабочих элементов) в рабочей группе и определяется как $\log_2(MW)+1$. Поскольку для платформы Intel MW=256, то для неё PW=9; а для платформы NVidia MW=1024 и следовательно PW=11. Вот этим и объясняется ограниченность варианта D.

Дополнить же этот вариант ещё одной стадией с циклическим вызовом процедуры fft_btfly (как в вариантах C или C^L) не представляется возможным, так как процедура ядра

fft_btflyLoopTL рассчитана на работу лишь при соблюдении условия: WS≥R, означающего, что количество пар (R) элементов в каждой группе не должно превосходить количества исполнителей (WS), имеющихся в каждой рабочей группе.

В целях дальнейшей оптимизации операции БПФ при последующем развитии OpenCL-программы попробуем снять это ограничение, а также будем активно использовать имеющуюся у каждого обрабатывающего элемента свою собственную внутреннюю память.

8. Активное использование внутренней памяти

Кроме доступа к глобальной памяти OpenCL-устройства и локальной памяти рабочей группы каждый обрабатывающий элемент может использовать ещё и свою внутреннюю или личную (private) память. Конечно, объём этой памяти очень небольшой, но зато обращение к ней выполняется намного быстрее, чем к локальной (и тем более к глобальной) памяти.

Попробуем выгодно использовать такую память для хранения отдельных участков обрабатываемого вектора, чтобы ещё более ускорить операцию БПФ. Для этого изменим OpenCL-программу варианта D следующим образом.

Заметим, что на начальных этапах 2-й части БПФ обрабатываются пары соседних элементов, взятые из непрерывного участка вектора. Значит, каждому обрабатывающему элементу можно поручить исполнить все операции БФ на нескольких **PA** первых этапах со всеми парами элементов назначенного ему участка длиной **NA**=$2^{\mathbf{PA}}$, сохраняя при этом эти элементы в своей личной памяти для ускорения доступа к ним.

Чтобы осуществить это, добавим в алгоритм 2-й части БПФ предварительную (нулевую) стадию (помимо уже рассмотренных двух), на которой будем запускать такую процедуру ядра:

```
#define MAX_NA 16
kernel void fft_btflyLoopA(int n,
 __global complex *X,
 __global complex *W, int PA) {
uint i,S,A,t,s,h,G,r,d,u,k,j,a,b;
complex XA[MAX_NA]; complex V;
i= get_global_id(0);// глобальный номер ОЭ
S=1<<PA;// S=2^PA размер обрабат. участка вектора
A=i*S; //A= его начальный индекс в векторе X
s=1;h=2;G=1;d=n/2;r=S/2; V=W[0];
if(PA==1){ // это если только один этап
 for(j=0,a=0;j<r;j++,a=a+h)
 {b=a+s; BTF(X[A+a],X[A+b],V);}
}else{
 for(j=0,a=0;j<r;j++,a=a+h){ b=a+s;
  BTFLY(X[A+a],X[A+b],V,XA[a],XA[b]);}
 for(t=2;t<Z;t++){
  for(k=0,u=0;k<G;k++,u=u+d){ V=W[u];
   for(j=0,a=k;j<r;j++,a=a+h)
    {b=a+s;BTF(XA[a],XA[b],V);}
  }//for k
  h=h*2;s=s*2;G=G*2;r=r/2;d=d/2;
 }//for t
 for(k=0,u=0;k<G;k++,u=u+d){ V=W[u];
  for(j=0,a=k;j<r;j++,a=a+h){ b=a+s;
  BTFLY(XA[a],XA[b],V,X[A+a],X[A+b]);}
 }//for k
}//if PA==1
}//fft_btflyLoopA
```

Выполняя эту процедуру, каждый параллельный исполнитель самостоятельно осуществляет первые **PA** этапов операций БФ со всеми парами элементов на выделенном ему участке вектора. Алгоритм этой процедуры, в основном, заимствует алгоритм 2-ой части функции FFT (см. п.2). Он отличается от него лишь тем, откуда читаются элементы вектора, над которыми выполняется операция БФ, и куда помещаются затем её результаты. На 1-ом этапе такие элементы берутся из глобальной памяти, а результаты помещаются в личную память. На последнем, наоборот, берутся из личной памяти, а результаты сохраняются в глобальной памяти. А на всех промежуточных этапах обрабатываются пары элементов, хранящиеся в личной памяти.

Для запуска этой процедуры ядра внесём необходимые изменения в OpenCL-программу.
Прежде всего, требуется получить дескриптор новой процедуры ядра:

```
kn5= clCreateKernel(prgrm,"fft_btflyLoopA",NULL);
```

Перед запуском назначим этой процедуре параметры (в 4-ом пункте функции clFFT):

```
#define NA 16
cl_uint PA=iLog2(NA);
clSetKernelArg(kn5,0,szI,&N);
clSetKernelArg(kn5,1,szM,&objX);
clSetKernelArg(kn5,2,szM,&objW);
clSetKernelArg(kn5,3,szI,&PA);
```

Размер участков, обрабатываемых на этой нулевой стадии, зафиксируем заранее (**NA**=16).

Чтобы запускать такую процедуру ядра на нулевой стадии БПФ, добавим в начало 5-го пункта функции clFFT вот такой фрагмент:

```
{ 0-ая стадия } // t=1,..,PA; PA= log2(NA);
gWS[0]=N/NA;// кол-во парал. исполнителей
clEnqueueNDRangeKernel(cmndQ,
 kn5,1,NULL,gWS,0,0,NULL,&ev5);
clWaitForEvents(1,&ev5);
```

Необходимо также немного изменить запускаемую на 1-ой стадии процедуру ядра. Ведь теперь, после исполнения PA этапов нулевой стадии, работа 1-ой стадии должна начинаться не с 1-го этапа, а с этапа PA+1. Поэтому вместо одного параметра PB поместим в заголовок процедуры два параметра (P1 и P2), а в теле процедуры поменяем некоторые операторы, которые зависят от этих параметров (см. фрагменты, помеченные жирным шрифтом):

```
kernel void fft_btflyLoopEL
(int n,__global complex *X,
 __global complex *W, int P1, int P2,
 __local complex *XL ) {
uint G,R,WS,k,j,s,h,a,b,u;
uint g,i,kG,jG, kl,jl,al,bl;
i=get_local_id(0);// локальный номер исполнителя
WS=get_local_size(0);// размер раб.группы
g=get_group_id(0);// номер раб.группы
G=1<<(P1-1); R=n>>P1;
kG=0;jG=g*WS/G;
get_GRkjuabL(P1); a=k+j*h; b=a+s;
if(P1==P2){BTF(X[a],X[b],W[u]);}else{
BTFLY(X[a],X[b],W[u],XL[al],XL[bl]);
for(t=P1+1;t<=P2-1;t++){
 barrier(CLK_LOCAL_MEM_FENCE);
 get_GRkjuabL(t);
 BTF(XL[al],XL[bl],W[u]);
}//for t
 barrier(CLK_LOCAL_MEM_FENCE);
 get_GRkjuabL(P2); a=k+j*h;b=a+s;
 BTFLY(XL[al],XL[bl],W[u],X[a],X[b]);
}//if P1==P2
}//fft_btflyLoopEL
```

Потребуется сделать изменения и в OpenCL-программе. Сменим имя процедуры ядра:

```
kn3=clCreateKernel(prgrm,"fft_btflyLoopEL",NULL);
```

И фрагмент назначения ей параметров:

```
cl_uint PA1=PA+1;
clSetKernelArg(kn3,3,szI,&PA1);
clSetKernelArg(kn3,4,szI,&PB);
clSetKernelArg(kn3,5,szC*NB,NULL);
```

В результате получим вариант **E** OpenCL-программы. Проверим, удаётся ли таким вариантом ускорить операцию БПФ и насколько.

В таблице 7 представлены коэффициенты ускорения, полученные в результате прогонов этого варианта на платформах Intel и NVidia.

Таблица 7. Ускорение операции БПФ OpenCL-программой варианта **E** для платформ Intel и NVidia

P	N=2^P	Intel		NVidia	
		K2	K2^	K2	K2^
12	4096	28.68	1.12	67.67	1.09
13	8192	50.16	1.17	130.64	1.01
14	16384	69.25	1.19	228.76	1.13
15	32768	77.95	1.02	332.92	1.11
16	65536	73.17	0.98	472.52	1.12
17	131072	72.42	0.96	394.59	1.35
18	262144	53.79	0.80	367.04	1.46
19	524288	---	---	443.02	1.46
20	1048576	---	---	443.84	1.46
21	2097152	---	---	455.04	1.47
22	4194304	---	---	443.23	1.43

Как видно из этой таблицы, на платформе NVidia вариант E, увы, почему-то совсем не улучшает коэффициент «чистого» ускорения (K2) операции БПФ, полученный вариантом D. И на платформе Intel ожидаемого улучшения тоже не наблюдается. Казалось бы, почему?

А всё дело в том, что из-за введения нулевой стадии приходится сокращать количество этапов, обрабатываемых на 1-ой стадии (t=PA+1,..,PB). А охватить 1-ой стадией следующие PB этапов (t=PA+1,..,PA+PB) не удаётся по той причине, что процедура ядра fft_btflyLoopEL (как и fft_btflyLoopCL) настроена на обработку лишь непрерывного участка вектора длиной **NB**= 2^{PB} элементов.

Чтобы избавиться от этого ограничения, надо усовершенствовать процедуру ядра, используемую исполнителями рабочей группы. Требуется сделать её универсальной, чтобы она могла обрабатывать требуемое количество этапов (t=P1,..,P2) на любом произвольно заданном участке вектора, элементы которого следуют друг за другом с некоторым фиксированным и не обязательно единичным шагом.

9. Универсальная процедура ядра для операции БПФ

Представленная ниже процедура ядра (fft_btflyLoopUL) позволяет на любых этапах (заданными параметрами P1 и P2) исполнять операции БФ с парами элементов внутри своей рабочей группы при различных соотношениях между WS и R: как при условии WS≥R, так и при условии WS<R. Схема назначения пар элементов исполнителям и алгоритм вычисления параметров назначаемой пары, применённый в этой процедуре, поясняются на рисунке 3.

В первом случае (WS≥R, см. на рис. 3 слева) исполнителям одной рабочей группы предстоит совместно обрабатывать несколько групп элементов, каждая из которых сосредоточена в отдельном столбце воображаемой матрицы, размещённой на месте заданного вектора по строкам размером по **sG** элементов (**sG**=2^{P1-1}).

А во втором случае (WS<R, см. на рис. 3 справа) столбец каждой такой группы элементов разделяется на несколько (**L=R/WS**) частей для того, чтобы обработку каждой из них можно было осуществить отдельной рабочей группой исполнителей.

С такой процедурой можно разбить обработку всех P этапов 2-ой части операции БПФ на 4 стадии и выполнять её по следующему алгоритму. На 0-ой стадии вызывать процедуру fft_btflyLoopA для исполнения первых **PA** этапов. Далее на 1-ой и 2-й стадии вызывать процедуру fft_btflyLoopUL, чтобы исполнить следующие **PB** этапов и **PC** этапов для обработки участков из **NB**=2^{PB} и **NC**=2^{PC} элементов соответственно. А на заключительной стадии опять-таки циклически вызывать процедуру fft_btfly. Причём количественное распределение этих стадий можно было бы варьировать, т.е. задавать

значения **PA**, **PB**, **PC** (или **NA**, **NB**, NC) как параметры при запуске OpenCL-программы.

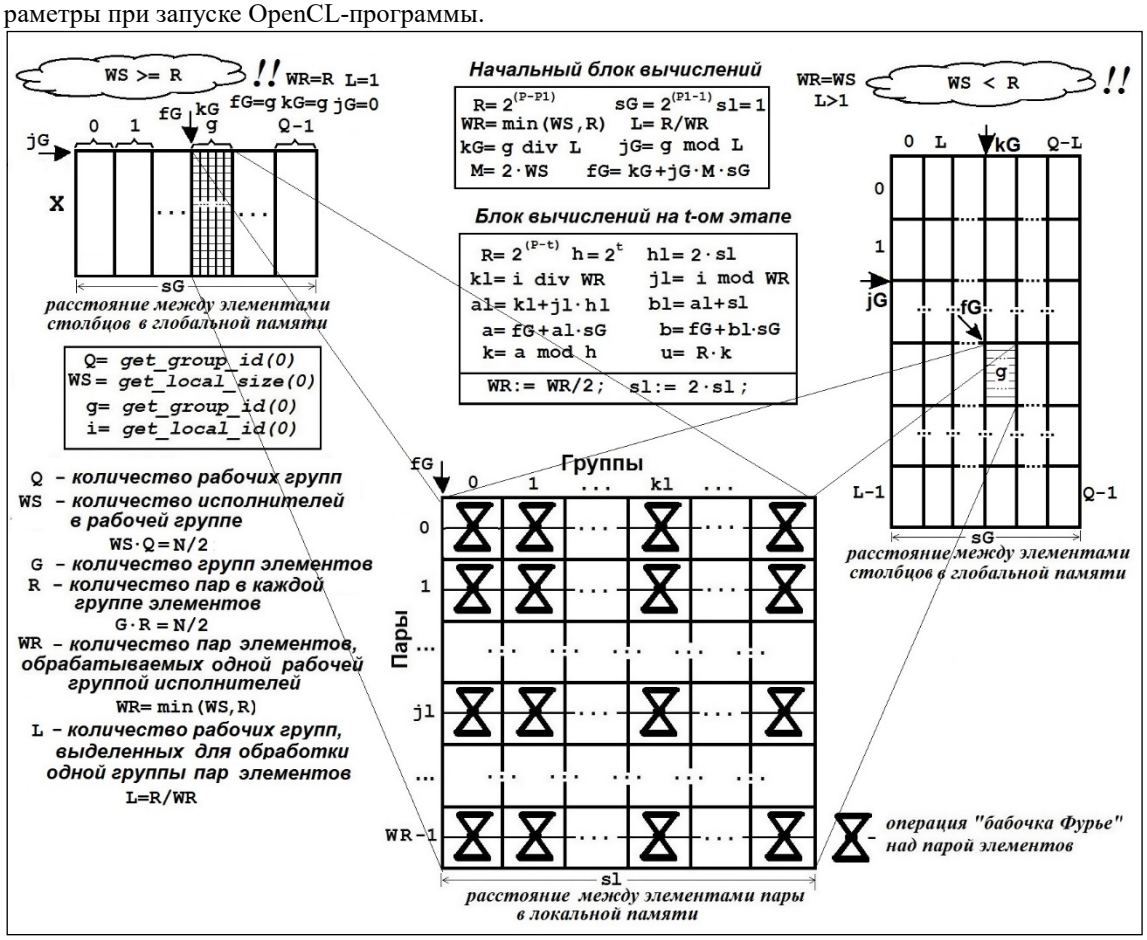

Рис. 3. Схема назначения пар в универсальной процедуре ядра 1-й и 2-й стадии варианта F

```
kernel void fft_btflyLoopUL
(int n,__global complex *X,
 __global complex *W, int P1,
 int P2, __local complex *XL ){
uint R,k,j,kl,jl,sl,h,a,b,al,bl,u;
uint g,i,WS,L,kG,jG,sG,fG;
i=get_local_id(0);// локальный номер исполнителя
WS=get_local_size(0);// размер раб.группы
g=get_group_id (0);// номер раб.группы
R=n>>P1; WR=MIN(WS,R);
L=R/WR;//L=к-во раб.групп для обработки X-группы
kG=g/L;jG=g%L; sG=1<<(P1-1);
fG=kG+jG*2*WS*sG;
sl=1; get_kjuabL(P1);
if(P1==P2){BTF(X[a],X[b],W[u]);}else{
BTFLY(X[a],X[b],W[u],XL[al],XL[bl]);
R=R>>1; WR=WR>>1; sl=2;
for(t=P1+1;t<=P2-1;t++){
 barrier(CLK_LOCAL_MEM_FENCE);
 get_kjuabL(t);
 BTF(XL[al],XL[bl],W[u]);
 R=R>>1; WR=WR>>1; sl=sl<<1;
}//for t
 barrier(CLK_LOCAL_MEM_FENCE);
 get_kjuabL(P2);
BTFLY(XL[al],XL[bl],W[u],X[a],X[b]);
}//if P1==P2
}//fft_btflyLoopUL
```

Для вычислений параметров пары элементов, назначаемой исполнителю на t-ом этапе, используется макроопределение **get_kjuabL**:

```
#define get_kjuabL(t) { \
 jl= i % WR; kl= i / WR; \
 al=kl+2*jl*sl; bl=al+sl; \
 a=fG+al*sG; b=fG+bl*sG; \
 k=a&((1<<t)-1);/*k=a mod h*/;u=R*k; }
```

Для осуществления такой обработки внесём необходимые изменения в OpenCL-программу:

```
kn3=clCreateKernel(prgrm,"fft_btflyLoopUL",NULL);
kn4=clCreateKernel(prgrm,"fft_btflyLoopUL",NULL);
```

Назначим процедуре ядра параметры её вызовов (в 4-м пункте действий функции clFFT):

```
cl_uint PB1=PA+1; cl_uint PB2=PA+PB;
cl_uint PC=Log2(NC);
clSetKernelArg(kn3,3,szI,&PB1);
clSetKernelArg(kn3,4,szI,&PB2);
clSetKernelArg(kn3,5,szC*NB,NULL);
cl_uint PC1=PB2+1; cl_uint PC2=PB2+PC;
clSetKernelArg(kn4,3,szI,&PC1);
clSetKernelArg(kn4,4,szI,&PC2);
```

```
clSetKernelArg(kn4,5,szC*NC,NULL);
```

И перепишем 5-й пункт списка действий функции clFFT, состоящий из 4-х стадий:

```
cl_uint t; cl_event ev2,ev3,ev4,ev5;
size t lWS[1]={NB/2}; gWS[0]=N/2;
{ 0-ая стадия } // t=1,..,PA; PA= log2(NA);
gWS[0]=N/NA;// кол-во парал. исполнителей
clEnqueueNDRangeKernel(cmndQ,
 kn5,1,NULL,gWS,0,0,NULL,&ev5);
clWaitForEvents(1,&ev5);
{ 1-ая стадия } // t=PA+1,..,PA+PB; PB= log2(NB);
clEnqueueNDRangeKernel(cmndQ,
 kn3,1,NULL,gWS,lWS,0,NULL,&ev3);
clWaitForEvents(1,&ev3);
{ 2-ая стадия } // t=PA+PB+1,..,PA+PB+PC;
lWS[0]= NC/2; // PC= log2(NC);
clEnqueueNDRangeKernel(cmndQ,
 kn4,1,NULL,gWS,lWS,0,NULL,&ev4);
clWaitForEvents(1,&ev4);
{ 3-ая стадия } // t=PA+PB+PC+1,..,P; P=log2(N);
for (t=PA+PB+PC+1; t<=P; t++) {
 clSetKernelArg(kn2,3,szI,&t);
 clEnqueueNDRangeKernel(cmndQ,
 kn2,1,NULL,gWS,NULL,0,NULL,&ev2);
 clWaitForEvents(1,&ev2);
}//for t
```

Полученный вариант OpenCL-программы обозначим как **F**. Обеспечиваемые им коэффициенты ускорения, полученные в результате прогонов на платформах Intel и NVidia, представлены в таблице 8.

Таблица 8. Ускорение операции БПФ OpenCL-программой варианта **F** для платформ Intel и NVidia

P	N=2^P	Intel		NVidia	
		K2	K2^	K2	K2^
8	256	2.15	1.11	7.55	1.64
9	512	4.80	1.23	12.48	1.82
10	1024	10.64	1.34	28.23	1.88
11	2048	20.41	1.42	55.81	1.75
12	4096	33.65	1.32	94.16	1.52
13	8192	49.87	1.17	169.69	1.31
14	16384	34.63	0.60	250.66	1.24
15	32768	48.03	0.63	275.00	0.92
16	65536	53.69	0.72	195.71	0.47
17	131072	58.22	0.77	197.15	0.67
18	262144	55.63	0.83	225.77	0.90
19	524288	49.45	0.95	319.08	1.05
20	1048576	**56.31**	**1.11**	358.84	1.46
21	2097152	52.56	1.11	395.74	1.28
22	4194304	**48.90**	**1.21**	415.87	1.34
23	2097152	**46.64**	**1.23**	**441.27**	**1.41**
24	4194304	**43.39**	**1.23**	**473.99**	**1.52**

Сравнивая их с коэффициентами предыдущих вариантов, можем сделать вывод, что этот вариант (**F**) предпочтительно применять лишь для векторов очень больших размеров: на платформе Intel он даёт выигрыш для векторов длиной $N \geq 2^{20}$, а на платформе NVidia − для векторов длиной $N \geq 2^{23}$.

10. Результаты оптимизации для платформ Intel и NVidia

Сравнив по производительности (по коэффициенту K2) все рассмотренные выше варианты OpenCL-программы выполнения операции БПФ на платформах Intel и NVidia, выберем для каждой длины самый оптимальный из вариантов и представим их все вместе в итоговых таблицах (см. таблицу 9 для платформы Intel и таблицу 10 для платформы NVidia).

Таблица 9. Результаты оптимизации OpenCL-программы операции БПФ для платформы Intel

P	N=2^P	V	K1	K2	K2^	K3	%K
8	256	$\mathbf{C^L}$	0.39	4.90	2.52	3.04	74.31
9	512	$\mathbf{C^L}$	0.87	10.05	**2.57**	**3.06**	75.78
10	1024	$\mathbf{C^L}$	1.16	17.90	2.25	2.56	79.94
11	2048	$\mathbf{C^L}$	2.18	29.76	2.07	2.29	82.55
12	4096	$\mathbf{C^L}$	3.74	45.93	1.80	1.95	84.03
13	8192	$\mathbf{C^L}$	6.22	64.29	1.51	1.58	87.13
14	16384	$\mathbf{C^L}$	11.38	78.70	1.35	1.40	87.58
15	32768	$\mathbf{C^L}$	17.83	**95.11**	1.25	1.28	**87.48**
16	65536	$\mathbf{C^L}$	30.43	85.08	1.14	1.18	70.99
17	131072	$\mathbf{C^L}$	40.91	83.57	1.11	1.14	72.26
18	262144	$\mathbf{C^L}$	45.39	72.70	1.08	1.12	65.03
19	524288	$\mathbf{C^L}$	41.71	54.94	1.06	1.10	**62.08**
20	104576	**F**	49.57	56.31	1.07	1.10	65.42
21	2097152	$\mathbf{C^L}$	47.02	52.95	1.12	1.19	69.22
22	4194304	**F**	45.12	48.90	1.21	1.31	69.51
23	8388608	**F**	44.24	46.64	**1.23**	1.34	67.85
24	16777216	**F**	41.52	43.39	**1.23**	1.36	64.35

Таблица 10. Результаты оптимизации OpenCL-программы операции БПФ для платформы NVidia

P	N=2^P	V	K1	K2	K2^	K3	%K
8	256	**D**	0.75	15.72	3.42	4.94	61.61
9	512	**D**	1.00	26.38	3.85	5.61	61.76
10	1024	**D**	2.00	56.74	3.78	5.15	66.76
11	2048	**D**	4.67	100.20	3.14	3.91	73.72
12	4096	$\mathbf{C^L}$	7.50	158.97	2.57	3.11	74.56
13	8192	$\mathbf{C^L}$	17.00	286.46	2.21	2.62	75.06
14	16384	$\mathbf{C^L}$	37.00	421.76	2.08	2.41	76.86
15	32768	$\mathbf{C^L}$	52.00	541.96	1.81	2.02	**79.70**
16	65536	$\mathbf{C^L}$	69.00	**666.94**	1.59	1.74	79.49
17	131072	**D**	150.00	545.52	1.86	2.35	73.79
18	262144	**D**	160.50	495.14	**1.96**	**2.72**	56.41
19	524288	**D**	209.00	**576.73**	1.90	2.66	**54.41**
20	1048576	**D**	222.00	566.46	1.87	2.53	55.42
21	2097152	**D**	240.88	562.78	1.82	2.36	60.01
22	4194304	**D**	**243.12**	521.69	1.68	2.05	65.00
23	8388608	**F**	235.79	441.27	1.41	1.57	72.08
24	16777216	**F**	257.80	473.99	**1.52**	1.73	71.58

Таким образом, применяя усовершенствованные варианты (B-F) OpenCL-программы, можно добиться более значительного ускорения операции БПФ в среде OpenCL. Как видно из приведённых таблиц, для векторов небольшой длины ($N \leq 2048$) удаётся увеличить коэффициент «чистого» ускорения (K2) примерно в 1,5-2

раза на платформе Intel и в 3-4 раза на платформе NVidia (см. столбец **K2^**). А для векторов очень большого размера ($N \geq 2^{22}$) – лишь примерно в 1,2 раза на платформе Intel и примерно в 1,5 раза на платформе NVidia.

Встаёт вопрос: а можно ли добиться ещё б**о**льшего ускорения? Чтобы на него ответить, вспомним, что операция БПФ складывается из двух частей: 1) бит-реверсной перестановки (копирования) элементов вектора; и 2) поэтапного цикла вычислений БФ («бабочек Фурье») над всевозможными парами элементов.

Все рассмотренные выше варианты (B-F) были нацелены лишь на ускорение 2-й части БПФ. Насколько удалось ускорить эту часть (по сравнению с вариантом A), как раз и показывает коэффициент **K3**, приведённый в таблицах 9-10. А значение из последнего столбца **(%K**) этих таблиц характеризует, какую долю (в процентах) эта часть занимает в общей длительности операции БПФ.

Чтобы прояснить, как влияет коэффициент **K3** и доля **%K** на итоговый коэффициент **K2**, решим следующую задачу.

Задача Операция **P** складывается из двух частей $\mathbf{P_1}$ и $\mathbf{P_2}$: $P=P_1+P_2$, для которых известны временн**ы**е доли их исполнения $\mathbf{d_1}$ и $\mathbf{d_2}$:

$$T=T_1+T_2,\ \mathbf{d_1}=T_1/T,\ \mathbf{d_2}=T_2/T.$$

Если ускорить исполнение части $\mathbf{P_1}$ в $\mathbf{u_1}$ раз, а части $\mathbf{P_2}$ — в $\mathbf{u_2}$ раз, то во сколько раз (**u**) ускорится исполнение всей операции **P**?

Решение Выразив новые длительности исполнения для P_1, P_2, P:
$T_1'=T_1/u_1=T\cdot d_1/u_1$, $T_2'=T_2/u_2=T\cdot d_2/u_2$, $T'=T_1'+T_2'$.
получим:
$T'=T\cdot d_1/u_1 + T\cdot d_2/u_2 = T\cdot(d_1\cdot u_2+ d_2\cdot u_1)/(u_1\cdot u_2)$.
Тогда коэффициент **u** ускорения операции P, равный отношению $u=T/T'$, должен вычисляться по формуле:

$$\mathbf{u= (u_1\cdot u_2) / (d_1\cdot u_2+ d_2\cdot u_1)} \quad \{\mathbf{Ф1}\}.$$

А в том случае, когда часть P_2 в новом исполнении исчезает совсем ($\mathbf{d_2}=0$ или $\mathbf{u_2}\to\infty$), вычисление коэффициента **u** можно упростить:
$T_2'=0$, $T'=T\cdot d_1/u_1$ и тогда $\mathbf{u= u_1/d_1}$ {**Ф2**}.
А при $\mathbf{u_1}=1$ это означает, что $\mathbf{u= 1/d_1}$ {**Ф3**}.

Вопрос: Какой можно ожидать итоговый выигрыш в ускорении всей операции, если добиваться ускорения лишь 2-й её части?

В качестве ответа на него приведём таблицу зависимости коэффициента ускорения **u** всей операции от коэффициента ускорения её 2-ой части $\mathbf{u_2}$ (при неизменном значении $\mathbf{u_1}=1$) для различных значений её доли $\mathbf{d_2}$ в общей операции (см. таблицу 11). В ней наглядно демонстрируется, что с ростом доли 1-ой части ($\mathbf{d_1}$) добиваться ускорения всей операции только за счёт ускорения её 2-й части становится всё проблематичней. И дальнейший рост коэффициента **u** упирается в «потолок»: его значение не сможет превысить 2.0 при $\mathbf{d_1}$=50%, 3⅓ − при $\mathbf{d_1}$=30%, 5.0 − при $\mathbf{d_1}$=20%, и 10.0 − при $\mathbf{d_1}$=10%.

Таблица 11. Зависимость общего коэффициента **u** от коэффициента $\mathbf{u_2}$ для разных значений $\mathbf{d_2}$ при $\mathbf{u_1}$=1

u= u_2 / ($d_1\cdot u_2+d_2$)					
u_2 \ d_1+d_2	**0,5+0,5**	**0,4+0,6**	**0,3+0,7**	**0,2+0,8**	**0,1+0,9**
2	1.33	1.43	1.54	1.66	1.82
3	1.5	1.67	1.875	2.14	2.5
4	1.6	1.82	2.11	2.5	3.08
5	1.67	1.92	2.27	2.77	3.57
6	1.71	2.0	2.4	3.0	4.0
7	1.75	2.06	2.5	3.18	4.375
8	1.77	2.11	2.58	3.33	4.71
9	1.8	2.14	2.65	3.46	5.0
10	1.82	2.17	2.7	3.57	5.26
20	1.9	2.33	2.985	4.16	6.9
50	1.96	2.43	3.185	27.78	8.47
100	1.98	2.46	3.26	4.8	9.17
$u_2\to\infty$	**2.0**	**2.5**	**3.33**	**5.0**	**10.0**

Анализируя таблицы 9 и 10, заметим, что после проведённой оптимизации 2-й части, доля 1-ой части БПФ становится весьма значительной: теперь она варьируется от 12.52% до 37.92% на платформе Intel и от 20.03% до 45.59% на платформе NVidia. А это значит, что дальнейшее ускорение операции БПФ в значительной степени зависит от того, удастся ли (и насколько!) ускорить 1-ю часть БПФ, т.е. бит-реверсную перестановку элементов вектора.

Но в среде параллельных исполнителей с глобальной памятью выполнить такую перестановку быстрее, увы, вряд ли возможно. Ведь в процедуре ядра fft_brvprst после вызова функции **k**=get_global_id(0) (для получения номера исполнителя) далее следуют лишь два оператора. Первый оператор m=BRT[k] получает значение бит-реверсного индекса из общей таблицы, а второй оператор X[m]=Y[k] просто копирует элемент одного вектора в другой. Каждый исполнитель при выполнении этой процедуры осуществляет 3 обращения в глобальную память. И сократить число таких обращений тут никак не получится.

Можно, конечно, отказаться от использования общей таблицы BRT и вычислять бит-реверсное значение в самой процедуре ядра (как это было описано, например, в [5,п.3.1]). Но это даст выигрыш лишь в том случае, когда такое вычисление будет осуществляться быстрее, чем чтение результирующего значения из общей таблицы, приготовленной в глобальной памяти. И такой вариант, вообще говоря, возможен, например, в случае наличия у процессорного ядра специальной команды, позволяющей для заданного целочисленного значения сразу же получить его бит-реверсное значение.

Но те платформы, на которых проводились

прогоны разработанных OpenCL-программ, такими процессорами с аппаратной поддержкой вычисления бит-реверсного значения, увы, не располагают. И поскольку ускорить 1-ую часть БПФ никак не удаётся, трудно рассчитывать на дальнейшее усовершенствование OpenCL-программы в целях ускорения операции БПФ.

11. Результаты оптимизации для платформы КОМДИВ

Проверим теперь, насколько позволяет применение описанных приёмов оптимизации повысить коэффициенты ускорения операции БПФ в среде OpenCL, обеспечиваемой в настоящее время микросхемой графического ускорителя семейства КОМДИВ. Все рассмотренные варианты (A-F) OpenCL-программы испытывались на аппаратной платформе, обеспечиваемой этой микросхемой, со следующими частотами: частота основного процессора (CPU) 400 Мгц, частота памяти 700 Мгц, частота системной шины 300 Мгц и частота графического процессора (GPU) 200 Мгц (Далее такую платформу будем сокращённо называть «платформа КОМДИВ».)

OpenCL-программа каждого варианта многократно прогонялась на такой платформе для векторов комплексных чисел (одинарной точности) различной длины $N=2^P$ (P=8,9,..,23). Полученные в результате этих прогонов усреднённые коэффициенты «чистого» ускорения (K2) операции БПФ различных вариантов представлены в таблице 12.

Таблица 12. Коэффициенты «чистого» ускорения (K2) операции БПФ различных вариантов OpenCL-программы, полученные на платформе КОМДИВ

P	$N=2^P$	A	C^L	D	E	F	K2!
8	256	0.07	**0.36**	0.35	0.25	0.25	**4.86**
9	512	0.15	0.53	0.49	0.41	**0.60**	**3.96**
10	1024	0.31	0.91	1.22	0.95	**1.31**	**4.21**
11	2048	0.63	1.51	2.25	1.87	**2.30**	**3.65**
12	4096	1.54	2.71	**3.75**	3.14	3.67	**2.44**
13	8192	3.6	4.77	**5.51**	4.93	4.55	**1.53**
14	16384	5.58	**6.40**	5.97	5.58	5.78	**1.15**
15	32768	7.81	**7.94**	6.59	6.31	6.99	**1.02**
16	65536	**10.47**	9.96	7.74	7.56	8.54	1.00
17	131072	**12.40**	11.54	---	---	8.99	1.00
18	262144	**12.63**	11.59	---	---	8.15	1.00
19	524288	**12.97**	11.87	---	---	8.42	1.00
20	1048576	**12.86**	11.79	---	---	8.62	1.00
21	2097152	**12.48**	11.48	---	---	8.97	1.00
22	4194304	**12.45**	11.54	---	---	9.38	1.00
23	8388608	**12.28**	10.91	---	---	8.97	1.00
K2! – наилучший коэффициент роста (K2^) среди всех вариантов = наибольший **K2** / **K2** варианта **A**							

Заметим, что на этой платформе разрешается объединять в рабочую группу не более 128 обрабатывающих элементов (MW=128), поэтому на ней вариант B можно применять только для векторов длиной не более 256 (2^8), а варианты D и E могут быть применены лишь для векторов длиной не более 65536 (2^{16}).

Как видно из этой таблицы, проведённая оптимизация позволила повысить коэффициент «чистого» ускорения (K2) операции БПФ на этой платформе лишь для векторов длиной не более 2^{15}(32768). Для векторов же б**о**льшего размера ($\geq 2^{16}$) такая оптимизация не дала ожидаемого эффекта, так что в итоге первоначально разработанный вариант OpenCL-программы оказался для них самым производительным.

Описанные приёмы оптимизации дают реальный практический эффект, когда память OpenCL-устройства действительно построена иерархически. Так что обращение к такой памяти для каждого обрабатывающего элемента подчиняется правилу: самая высокая скорость обеспечивается для доступа к своей внутренней памяти, затем, чуть помедленней – к локальной памяти его рабочей группы, а самая низкая – к глобальной общей памяти всего устройства.

Но микросхема графического ускорителя семейства КОМДИВ такую иерархию памяти OpenCL-устройства в настоящее время в полной мере не обеспечивает. Поэтому описанные приёмы оптимизации не получается эффективно применить (на этой платформе КОМДИВ) для обработки данных большого объёма.

12. Заключение

Полученный автором опыт разработки разнообразных вариантов программ, ускоряющих операцию быстрого преобразования Фурье в среде OpenCL, позволяет сделать следующий вывод. Для разработки эффективных программ по технологии OpenCL требуется не только изучить богатый арсенал её средств, но и освоить также приёмы оптимизации процедур ядра с учётом иерархического строения памяти устройств OpenCL среды.

Публикация выполнена в рамках государственного задания ФГУ ФНЦ НИИСИ РАН «Проведение фундаментальных научных исследований (47 ГП)» по теме № FNEF-2022-0004 «Разработка архитектуры, системных решений и методов для создания микропроцессорных ядер и коммуникационных средств семейства систем на кристалле двойного назначения», Рег. № 122041100063-0.

Optimization of Fast Fourier Transform in OpenCL Environment

A.A. Burtsev

Abstract. The article focuses on the use of OpenCL technology, which allows to use powerful GPU resources to improve the performance of computing programs. Various variants of parallel programs designed to accelerate fast Fourier transform operations in an OpenCL environment are supposed. Considered variants are compared in terms of performance in order to determine the most optimal for processing complex vectors of different lengths.

Keywords: parallel programming, OpenCL technology, heterogeneous systems, microprocessors of the KOMDIV family, fast Fourier transform

Литература

1. В.В. Воеводин, Вл.В. Воеводин. Параллельные вычисления. Спб., БХВ-Петербург, 2004.

2. Официальный сайт ФНЦ НИИСИ РАН. Разработка СБИС. Развитие микропроцессоров с архитектурой КОМДИВ, https://www.niisi.ru/devel.htm

3. Официальный OpenCL–сайт организации Khronos Group, http://www.khronos.org/opencl/

4. А.А. Бурцев. Оптимизация операции перемножения матриц на основе технологии OpenCL. «Труды НИИСИ РАН», Т. 10 (2020), № 5-6, 100–112.

5. А.А. Бурцев. Ускорение быстрого преобразования Фурье на основе технологии OpenCL. «Труды НИИСИ РАН», Т. 11 (2021), № 4, 27–37.

6. Стивен Смит. Цифровая обработка сигналов. Практическое руководство для инженеров и научных работников. М., Додека-XXI, 2012.