OpenMP to hStreams: A Compiler Framework for Automatic

Translation and Optimization

一、基本流程

1、代码分析：遍历代码的AST树，找到需要在加速器端执行的循环位置，确定hstreams API插入位置，确定kernel参数信息，确定需要创建及传输的memory buffer信息，确定#partitions和#tasks。

1. 定位到#pragma omp parallel for修饰的循环。
2. 分析循环的for语句，确定循环变量的初始化语句和结束判断语句，并推断出循环变量取值范围及循环的迭代次数。
3. 找出所有在循环中使用、在循环外声明且没有在循环中进行初始化的变量。对这些变量根据是否是指针类型进行分类，作为kernel函数的参数。对于那些在循环中进行了初始化的变量，作为kernel的局部变量，需要在kernel中为这些变量加入声明语句。
4. 识别出kernel中的所有访存操作，记录访存操作的数组名、index表达式、读/写内存操作。
5. 根据kernel中对数组的访问（读/写），判断是否需要将数组从host传到device，或从device传到host。
6. 根据数组的index表达式，判断数组能否进行分块传输。
7. 根据malloc函数或者数组的index取值范围，推断数组规模。数组可能被访问多次，存在多个index表达式，需要推断出这些index的取值范围并进行比较。
8. 根据kernel循环中指令数量、可以分块传输的数组个数等，判断是否需要将该kernel放在加速器上执行。
9. 如果需要加速，则将分析结果保存到WriteInFile类中，并调用相关函数生成host文件和kernel文件。

2、host文件生成：根据代码分析结果，读取源文件生成host文件。

1. 插入hStreams头文件包含语句。
2. 插入hStreams初始化代码。
3. 插入memory buffer创建语句，需要代码分析过程提供buffer变量名、类型、大小等信息。
4. 将不能分块传输的buffer先传到device上。
5. 创建子任务循环。需要确定子任务数量。
6. 在循环中加入buffer传输语句和kernel执行语句，并设置kernel参数。
7. 插入数据回传语句及同步语句。
8. 插入hStreams结束代码。
9. Kernel文件生成：根据代码分析结果，读取源代码生成kernel文件。
10. 插入hStreams头文件包含语句。
11. 创建kernel函数。需要代码分析过程提供kernel所有参数信息。
12. 在kernel函数中首先加入参数的类型转换语句。
13. 修改kernel循环的循环变量初始值和终值，使循环的迭代次数等于子任务的规模。

二、优化技术

1、优化重复数据传输。

以表1中的代码为例，在原代码中每个子任务之间的数据传输存在重复的数据，重复传输的数据量大小为2\*kernelN。优化后通过使用event保证子任务之间数据传输的顺序，可以避免传输重复数据。

表1：重复数据传输代码示例

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | for(int i=0;i<dataN; ){  xheader = i;  xtail = xheader +task;  eheader = xheader - kernelN;  etail = xtail + kernelN;  hStreams\_app\_xfer\_memory(  &h\_Data[eheader],  &h\_Data[eheader],  (etail-eheader) \* sizeof(float),  stream, HSTR\_SRC\_TO\_SINK,  NULL);  …………  stream = (stream+1)%MAXLOGSTR;  i = xtail;  } | for(int i=0;i<dataN; ){  xheader = i;  xtail = xheader +task;  eheader = xheader + kernelN;  etail = xtail + kernelN;  hStreams\_EventStreamWait(  stream, 1, &eventCpy[counter-1],  0, NULL, NULL);  hStreams\_app\_xfer\_memory(  &h\_Data[eheader],  &h\_Data[eheader],  (etail-eheader) \* sizeof(float),  stream, HSTR\_SRC\_TO\_SINK,  &eventCpy[counter]);  …………  stream = (stream+1)%MAXLOGSTR;  i = xtail;  counter++;  } |
|  | 原代码 | 优化后代码 |

对fwt程序进行优化后，相对单流的加速比从1.14提升到1.46。

2、优化内存访问。

以表2中的代码为例，由于任务划分只能划分最外层循环，优化前的代码中A、B数组不能分块传输，优化后可以将A数据分块传输，而且还提高了数据访问的局部性。

表2：内存访问优化代码示例

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #pragma omp parallel for collapse (2)  for (int mm = 0; mm < m; ++mm) {  for (int nn = 0; nn < n; ++nn) {  float c = 0.0f;  for (int i = 0; i < k; ++i) {  float a = A[mm + i \* lda];  float b = B[nn + i \* ldb];  c += a \* b;  }  C[mm+nn\*ldc] = C[mm+nn\*ldc] \*  beta + alpha \* c;  }  } | A = AT;  B = BT;  #pragma omp parallel for collapse (2)  for (int mm = 0; mm < m; ++mm) {  for (int nn = 0; nn < n; ++nn) {  float c = 0.0f;  for (int i = 0; i < k; ++i) {  float a = A[i + mm \* lda];  float b = B[i + nn \* ldb];  c += a \* b;  }  C[mm+nn\*ldc] = C[mm+nn\*ldc] \*  beta + alpha \* c;  }  } |
|  | 原代码 | 优化后代码 |

1. 待解决的问题
2. 对于间接索引的数组，无法推断出数组规模。例如下面代码中的数组h\_indices，其索引变量与h\_ptr数组中的值相关，从而无法推断出h\_indices索引变量的取值范围。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | for(k=0;k<bound;k++ ) {  int j = h\_ptr[k] + i;  int in = h\_indices[j];  float d = h\_data[j];  float t = h\_x\_vector[in];  sum += d\*t;  } |

1. 对于不同子任务之间数组访问存在overlap的情况，目前还不能推断出overlap的规模。例如下面代码，在最外层循环的不同迭代之间，对数组A0的访问存在overlap，在子任务执行前还需要将overlap的数据传到device端。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | for(i=1;i<nx-1;i++)  {  int j,k;  for(j=1;j<ny-1;j++)  {  for(k=1;k<nz-1;k++)  {  Anext[Index3D (nx, ny, i, j, k)] =  (A0[Index3D (nx, ny, i, j, k + 1)] +  A0[Index3D (nx, ny, i, j, k - 1)] +  A0[Index3D (nx, ny, i, j + 1, k)] +  A0[Index3D (nx, ny, i, j - 1, k)] +  A0[Index3D (nx, ny, i + 1, j, k)] +  A0[Index3D (nx, ny, i - 1, j, k)])\*c1  - A0[Index3D (nx, ny, i, j, k)]\*c0;  }  }  } |

1. 在目前的实现中，只要kernel对数组进行了读操作，就需要将数组从host传到device。但是在kernel循环中，可能会对数组进行初始化，然后再对数组进行读写操作。对于这种情况，不需要将数组从host传到device。例如下面代码中的数组sum。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | for (r = start\_index; r < end\_index; r++)  for (q = 0; q < \_PB\_NQ; q++) {  for (p = 0; p < \_PB\_NP; p++) {  sum[p] = SCALAR\_VAL(0.0);  for (s = 0; s < \_PB\_NP; s++)  sum[p] += A[r][q][s] \* C4[s][p];  }  for (p = 0; p < \_PB\_NP; p++)  A[r][q][p] = sum[p];  } |

1. 分析host端读取kernel中进行了写入操作的数组位置，将hStreams同步操作语句插入到该语句之前，从而减少同步时间。而如果host端后续没有对该数组的读取操作，则不需要将该数组从device传到host。由于涉及到对kernel循环外面的代码分析，目前尚未实现。
2. 其余创新点
   1. 使用回归方法/多标签方法，性能72-79%。
   2. 在task划分时，进行load balance优化，保证每个子任务的计算量很接近。Spmv程序。更高级的load balance是保证每个硬件线程的计算量很接近！待测试
   3. 对加速比超过2的程序进行了性能分析。Kernel性能的减少有两种可能，一是omp处于内层循环，二是减少降低首次迭代开销（首次迭代开销是其余迭代开销的3-几十倍）。B/p>1的意义是隐藏数据传输时间。各硬件线程上的负载平衡影响b和p的选择。分析两个程序的最优b、p的取值？
   4. 更新特征值，使用omp循环的迭代次数，最大子任务数？