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j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。如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？将分类变量k1转换成指示变量k1a和k1b，k3重新进行定义为omp循环迭代次数?MAX blocks重新定义为划分维度的那个循环的循环变量取值范围长度？Size ratio重新定义为传输的最大mem大小除以max blocks？Size ratio2定义为传输的全部mem 总大小除以max blocks？Kernel exe、h2d Transfer、Cache miss、branch miss除以max blocks作为特征值来区分不同程序？Kernel\_size定义为kernel函数体指令数量/函数执行时指令访问内存的footprint？Ins\_issue定义为Kernel执行时流出指令数？Load\_stmt定义为kernel函数体中访存语句数量，s\_load\_ins定义为kernel执行时发出的标量load指令数量，v\_load\_ins定义为kernel执行时向量load指令数量。Save\_stmt，s\_save\_ins,v\_save\_ins类似定义。
   5. 交叉验证留一法是否合理？测试集与训练集的特征值分布是否相同？不同的话是否说明交叉验证无效？
   6. 对特征值进行PCA分析，分析不同变量之间是否存在关系
   7. 对于回归预测，分析各特征值与预测值的t-检验及F检验，分析预测值与特征值之间是否存在线性关系。要注意多重共线性问题，即特征值之间存在线性关系的话会影响回归模型。对p-b进行性能排序，根据p-b的顺序对程序-输入进行分类，根据p-b占最高性能的比值的倒数倍/取对数作为权重？再分析不同类的各个特征值是否具有区分度。可以根据特定的p-b性能对程序-输入进行分类。
   8. 在给定特征值时，预测值存在一个置信区间，是否可以用来修改预测结果？比较前几个预测加速比相近的p-b组合，最后挑选出实际性能最好的一个？预测值不应该用加速比，而是以最高性能为基准的减速比，或再对减速比取倒数/对数？
   9. 用F检验对每个特征值进行筛选。
   10. 分析特征值分布情况？正态化？标准化？
3. 代码分析

涉及到的类：CommonOptionsParser, ClangTool, MyASTVisitor, MyASTConsumer, MyFrontendAction

CommonOptionsParser：Clang tools常用选项的parser，可以解析出头文件路径、源代码文件、编译选项等。

ClangTool：管理源代码文件列表，为每个源文件分别调用FrontendAction。

MyFrontendAction：提供创建ASTConsumer类实例的函数。提供文件处理前/后的回调函数。

MyASTConsumer：提供处理clang pase源代码后生成的AST 树中的最高层声明语句的函数HandleTopLevelDecl，在该函数中对每个TopLevelDecl调用ASTVisitor对象的TraverseDecl函数进行访问。

MyASTVisitor：对整个AST树进行深度优先遍历，并访问每个结点。

MyASTVisitor类的成员函数：

find\_usage(Expr \*, ValueDecl \*): 如果表达式中使用了该变量，则返回True，否则返回False。

analysis\_declref(DeclRefExpr \*, string &, string &)：评估DeclRefExpr所表示变量的值范围，最大值最小值以字符串形式表示。返回值为3,将数组索引中的所有变量的评估值设为3,用来进行比较索引值大小，从而对数组最大访问范围进行推断。

analysis\_paren(Expr\*, string&, string&):处理括号表达式，因为括号可能会存在嵌套，所以需要递归处理。评估括号内表达式的值范围，最大最小值以字符串形式表示，返回括号内表达式的评估值。

analysis\_bin\_op(BinaryOperator \*, string& string&):评估二元操作表达式值范围，最大最小值以字符串形式表示，返回该表达式的评估值。

analysis\_index(Expr\*, string&, string&):评估表达式的取值范围，最大最小值以字符串形式表示，返回该表达式的评估值。

evalue\_index(Expr\*, struct var\_data \*):评估索引表达式的取值范围及评估值并保存在var\_data中。struct var\_data是数组变量，expr是数组的索引表达式。因此数组变量可能被访问多次，存在多个索引表达式，因此会比较当前索引表达式的评估值与var\_data中保存的之前索引表达式的评估值，如果大于value\_max，则更新var\_data的max\_value\_str（最大取值范围）和value\_max（最大评估值）。如果小于value\_min，则更新var\_data的min\_value\_str（最小取值范围）和value\_min（最小评估值）。

analysis\_array(ArraySubscriptExpr\*, string&, vector<Expr\*>&): 函数提取表达式中数组变量名(字符串形式)，及索引表达式(Expr\*的向量)。ArraySubscriptExpr是数据访问表达式，如A[1][2]。在AST中多维数组表示为嵌套的ArraySubscriptExpr，因此函数进行了递归调用。

check\_mem\_access(BinaryOperator\*,string&,vector<Expr\*>&,string&,vector<Expr\*>&):检查二元操作表达式是否有访存操作。返回0表示是赋值操作，1表示其他，2表示\*=/+=等复合赋值操作。如果右值表达式是数组访问，则提取出数组名及索引变量；如果是非指针类型的变量，则记录变量名；如果是访存操作，同样提取出数组名及索引变量。左值表达式进行相同处理。

get\_str(Stmt \*):返回语句的字符串形式。主要用于在评估表达式取值范围时使用，对于有的表达式目前没有进行解析，则会用该表达式的字符串形式表示其最大最小取值范围。

query\_var(string, struct var\_data \*\*):查询变量的作用域，并通过参数返回变量的var\_data指针。返回0表示变量是在kernel内部定义的，返回1表示变量是在kernel外部定义。MyASTVisitor类维护了一个vector<struct Scope\_dat>对象，记录了每个作用域中定义的变量。本函数通过该对象查询变量的作用域。

Insert\_var\_data(VarDecl \*, struct Scope\_data&):为声明的变量创建var\_data对象，并保存到当前作用域Scope\_data对象中。如果当前作用域中没有该变量的记录，则会创建var\_data对象，并初始化var\_data对象，包括记录变量名，变量的声明语句，变量类型。如果变量是指针类型，还会记录变量所指向元素的类型大小。

clean\_kernel\_info():初始化类的struct Kernel\_Info对象。为多kernel程序做准备。类的struct Kernel\_Info对象仅仅是在对kernel信息进行分析时临时保存信息用，分析完毕后会将kernel信息保存到类的vector<struct Kernel\_Info>对象中，此时需要初始化临时保存kernel信息的对象。

MyASTVisitor(vector<struct Kernel\_Info>&, vector<struct Scope\_data>&):MyASTVisitor类的构造函数。初始化process\_state为0,表示目前处理的代码是非kernel代码。

Initialize(ASTContext&):初始化类成员变量Ctx指向传入的ASTContext对象，初始化化SourceManager \*对象SM指向ASTContext的源代码管理器。

dataTraverseStmtPre(Stmt \*):在访问Stmt对象前，本函数会被调用。本函数对作用域对象进行维护。遇到新的作用域，如For语句、括号等，创建新的作用域对象，并在作用域对象中记录Stmt类型。当遇到括号类型时，如果当前作用域中有函数的声明，则认为正进入函数体中，记录函数体第一行和最后一行的行号作为hstream初始化及清扫代码的插入位置，并将函数参数变量加入到作用域中。另外如果进入到OMPParallelForDirective指导的循环中，则将process\_state置为1表明进入到kernel代码（这里将每个omp for循环视为一个kernel候选）。

dataTraverseStmtPost(Stmt\*):在访问Stmt对象后，会调用本函数。首先将当前作用域从栈中弹出。如果退出的是omp for循环，则视为该kernel到此结束，开始分析kernel中访问的所有数组变量，并为其创建mem\_xfer对象记录数组大小、类型、创建buffer的位置、传输代码的位置、传输方向等信息。然后将kernel\_info对象压入到栈中，清空kernel\_info对象，将process\_state置为2,表明位于kernel代码后（预留为优化用）。

VisitStmt(Stmt\*):访问Stmt对象时调用。如果是ForStmt语句且process\_state等于1（即kernel代码中的for语句），记录循环变量在循环迭代过程中的最小值、最大值（目前仅仅处理步长为1的情况）。如果该ForStmt语句是omp for指导语句修饰的，记录该循环在源代码中的起始与结束行号（两者之间的代码就是kernel函数体），及循环变量的初始值和总的迭代次数（任务划分时使用）。如果是kernel内的DeclRefExpr语句，而且对应的变量是在kernel范围外声名，则记录该变量作为kernel参数。如果是BinaryOperator表达式，分情况进行处理：统计kernel中计算操作数量（预留性能评估用）；如果非指针变量在kernel中被初始化，则将该变量作为kernel的局部参数（不需要传值）；如果是malloc函数调用，则记录对应指针变量分配的内存空间大小；对于kernel内部的访存操作，判断是读或写内存操作，确定需要将指针变量所指向的内存数据从host传到device还是从device传回host。

VisitVarDecl(VarDecl \*):处理变量声明语句，将新声明的变量插入到作用域对象。对于kernel内的变量声明还需要判断其初始化表达式是否包括访存操作。

VisitFunctioDecl(FunctionDecl \*):处理函数声明语句，如果该函数在声明时有函数体，则在作用域中记录该函数（用于表明下个作用域就是该函数的函数体）。

newFrontendActionFactory<MyFrontendAction>().get()返回的是clang::tooling::FrontendActionFactory对象。

**ClangTool.run**()执行过程：处理compilation database，生成编译选项。遍历所有输入文件，为每个文件创建一个ToolInvocation对象，参数是编译选项，ToolAction和文件。然后执行ToolInvocation.run()方法。

ToolInvocation.run(): 根据编译选项创建clang::driver::Compilation对象，再由Compilation对象创建llvm::opt::ArgStringList对象，再由ArgStringList对象创建clang::CompilerInvocation对象。然后调用runInvocation函数，执行ToolAction.runInvocation()方法，使用CompilerInvocation对象和作为参数。

FrontendActionFactory::runInvocation()：创建clang::CompilerInstance对象。然后执行CompilerInstance::ExecuteAction()函数。

CompilerInstance::ExecuteAction()：根据目标调整语言选项，执行FrontendAction的BeginSourceFile、Execute和EndSourceFile三个函数。

FrontendAction.BeginSourceFile()：这里执行的是FrontendAction::BeginSourceFile函数。会调用CI.createPreprocessor来对源文件进行预处理。创建AST Context，AST Consumer（此时会调用MyFrontendAction中的CreateASTConsumer函数来创建）等。

FrontendAction::Execute()：会调用ParseAST()函数，在该函数内遍历全部TopLevelDecl，并调用Consumer->HandleTopLevelDecl()来处理。

Consumer->HandleTopLevelDecl()：对传进来的所有Decl，调用Visitor.TraverseDecl()函数。

clang::RecursiveASTVisitor<MyASTVisitor>::TraverseDecl()：这是一个模板函数（/home/moon/local/include/clang/AST/RecursiveASTVisitor.h:697）。根据Decl的类型分别进行处理。

1. 生成代码模板

Kernel文件生成：

|  |
| --- |
| 原#include语句  #include <intel-coi/sink/COIPipeline\_sink.h>  COINATIVELIBEXPORT void  kernel (uint64\_t arg0,  uint64\_t arg1,  ……  )  {  Int start\_index = (int) arg0;  Int end\_index = (int) arg1;  Other parameters……  #pragma omp parallel for  for (var = start\_index; var < end\_index; )  {  原代码  }  } |

Host文件生成：

|  |
| --- |
| #include <hStreams\_source.h>  #include <hStreams\_app\_api.h>  #include <intel-coi/common/COIMacros\_common.h>  ……  Function() {  uint32\_t logical\_streams\_per\_place= 1;  uint32\_t places\_per\_domain = 2;  HSTR\_OPTIONS hstreams\_options;  hStreams\_GetCurrentOptions(&hstreams\_options, sizeof(hstreams\_options));  hstreams\_options.verbose = 0;  hstreams\_options.phys\_domains\_limit = 256;  char \*libNames[20] = {NULL,NULL};  unsigned int libNameCnt = 0;  libNames[libNameCnt++] = "kernel.so";  hstreams\_options.libNames = libNames;  hstreams\_options.libNameCnt = (uint16\_t)libNameCnt;  hStreams\_SetOptions(&hstreams\_options);  int iret = hStreams\_app\_init(places\_per\_domain, logical\_streams\_per\_place);  if( iret != 0 )  {  printf("hstreams\_app\_init failed!\n");  exit(-1);  }  (hStreams\_app\_create\_buf((type) var, size));  ……  (hStreams\_app\_xfer\_memory((type) var, (type) var, size, 0, HSTR\_SRC\_TO\_SINK, NULL));  ……  int sub\_blocks = len/task\_blocks;  int remain\_index = len%task\_blocks;  int start\_index = 0;  int end\_index = 0;  uint64\_t args[];  args[] = (uint64\_t) var;  hStreams\_ThreadSynchronize();  start\_index = value;  for (int idx\_subtask = 0; idx\_subtask < task\_blocks; idx\_subtask ++)  {  args[0] = (uint64\_t) start\_index;  end\_index = start\_index + sub\_blocks;  if (idx\_subtask < remain\_index)  end\_index ++;  args[1] = (uint64\_t) end\_index;  (hStreams\_app\_xfer\_memory(&var[start\_index], &var[start\_index], (end\_index - start\_index) \* sizeof (type), idx\_subtask % logical\_streams, HSTR\_SRC\_TO\_SINK, NULL));  ……  hStreams\_EnqueueCompute(  idx\_subtask % logical\_streams,  kernel,  val\_num,  pointer\_num,  args,  NULL,NULL,0));  (hStreams\_app\_xfer\_memory(&var[start\_index], &var[start\_index], (end\_index - start\_index) \* sizeof (type), idx\_subtask % logical\_streams, HSTR\_SINK\_TO\_SRC, NULL));  ……  start\_index = end\_index;  }  hStreams\_ThreadSynchronize();  (hStreams\_app\_xfer\_memory((type)var, (type)var, size, 0, HSTR\_SINK\_TO\_SRC, NULL));  ……  hStreams\_ThreadSynchronize();  ……  ……  hStreams\_app\_fini();  } |