IBM Journal of Research and Development

IEEE Xplore® Digital Library
http://ieeexplore.ieee.org/Xplore

OpenMP 4.5 Compiler Optimization for GPU Offloading

E. Tiotto, B. Mahjour, W. Tsang, X. Xue, T. Islam, W. Chen

Ability to efficiently offload computational workloads to GPUs is critical for the success of hybrid CPU-GPU architectures, such as the Summit and Sierra supercomputing systems. OpenMP 4.5 is a high-level programming model which enables the development of architecture and accelerator independent applications. This paper describes aspects of the OpenMP implementation in the IBM XL C/C++ and XL Fortran OpenMP compilers which aid programmer achieve performance objectives. This includes an inter-procedural static analysis the XL optimizer uses to specialize code generation of the OpenMP distribute parallel do loop within the dynamic context of a target region, and other compiler optimizations designed to reduce the overhead of data transferred to an offloaded target region. We introduce the heuristic used at runtime to select optimal grid sizes for offloaded target team constructs. These tuned heuristics lead to an average improvement of 2X in the runtime of several target regions in the SPEC ACCEL V1.2 benchmark suite. In addition to performance enhancement this paper also presents an advanced diagnostic feature implemented in the XL Fortran compiler to aid in debugging OpenMP applications offloaded to accelerators.

Introduction

Recent supercomputers are increasingly relying on hybrid system architectures and the computational power of general-purpose energy-efficient graphic processing units (GPUs) to achieve high-performance and scalability goals. As a case in point we consider the November 2018 and June 2019 TOP500 number one and two supercomputers, the Summit at Oak Ridge National Laboratory (ORNL) and Sierra at Lawrence Livermore National Laboratory (LLNL). The building blocks of both supercomputers is the IBM AC922 server node [AC922], containing two IBM POWER9® multicore CPUs and up to six NVIDIA Tesla® V100 GPUs, coupled to the CPUs by an NVLINK™ 2.0 interconnect.

A key challenge application programmers face to efficiently use these hybrid computing environments is the exploitation of both the CPUs and GPUs available on a node, as well as scaling their applications across the cluster nodes. Programming the GPUs is a non-trivial process that traditionally requires adopting low-level programming models such as the CUDA C/C++ [CudaC], CUDA Fortran [CudaF], or OpenCL [OpenCL].

To increase application portability as well as programmer productivity the use of higher-level abstractions is highly desirable. Programming model abstractions based on directives are particularly well suited to exploit coarse-level parallelism found at the loop nest level in many scientific programs. Today, amongst directive based parallel programming models, OpenMP [OpenMP] is arguably the most widely adopted and is traditionally very well supported on many SMP systems.

Since version 4.0, the OpenMP API specification has been extended to support accelerators such as those found on the AC922 nodes. The introduction of the OpenMP target constructs allows programmers to offload units of computation to an accelerator, in an architecture agnostic fashion, thereby freeing them from the burden of low level device specific enablement and performance tuning of their code for the accelerator architecture at hand.

It is then the prerogative of an optimizing compiler to translate the code within the target region, as specified in various directives, into optimized platform tuned code. The translation process entails managing allocation and deallocation of data structures used to transfer data between system memory and GPU global memory, as well as data movements, synchronization and execution of the computation on the GPU.

Efficient translation of work-sharing loops present within the dynamic scope of a target directive is key to take full advantage of the "Single Program, Multiple Data" (SPMD) capabilities of modern GPU architectures, and to achieve the low runtime overhead necessary to amortize the cost of data communication between device and host system when offloading computation to the accelerator.

In this paper we will present techniques explored and used in the IBM XL C/C++ and Fortran production compilers [IBMXL] to efficiently map key OpenMP device language constructs to NVIDIA GPUs, transformations used to reduce the overhead of data transfers to and from the GPU, and the heuristic used at runtime to select the grid size (i.e. number of blocks, threads per block) used to execute OpenMP target regions on the target GPU.

The remainder of this paper is organized as follows:

Section 1 presents an inter-procedural static compiler analysis geared toward identifying opportunities for SPMD specialization of the code generated for the commonly used OpenMP distribute parallel do loop in a target team context.

Section 2 describes a device function devirtualization technique used to optimize the code generated for a parallel region in a target region offloaded to the GPU.

Digital Object Identifier 10.1147/JRD.2019.2962428

Section 3 presents transformations targeting optimization of firstprivate variables and the elimination of unused map clauses in a target region, resulting in reduction of data communication between host and device.

Section 4 presents the algorithm and heuristic used by the XL OpenMP runtime system to select an efficient number of teams and threads per team when offloading a *target team* region to the GPU.

Finally, in Section 5 we describe an advanced diagnostic feature implemented in the XL Fortran compiler to report some hard to discover semantic errors in OpenMP device constructs.

1 SPMD code generation for the distribute parallel loop constructs in a target team region

The OpenMP API specification [OpenMP] allows programmers to offload regions of code (called *target regions*) to an accelerator. A target region may contain other OpenMP directives including, but not limited to, parallel regions (possibly nested) as well as sequential code. Although the implementation needs to support the full generality of directives that may be placed inside a target region, it is common in practice to have *target team* regions that contain, directly or indirectly, a *distribute parallel* loop. In this section we will describe the analysis and code generation techniques the XL compiler high-level optimizer employs to specialize such commonly used construct, along with a performance comparison between the specialized and the generic code generation schemes.

In order to introduce this optimization, we will first illustrate the generic code the compiler generates for the distribute parallel loop construct with a simple example (Figure 1). We will use this example throughout this section with some small modifications.

Figure 1: OpenMP stream triad microkernel

```
!$omp target teams distribute parallel do
     map(from: a) map(to: b,c)
do j = 1,n
  a(j) = b(j) + scalar * c(j)
end do
```

The generic pseudo code generated for the GPU kernel corresponding to the *target team* region in the example is in Figure 2a. The code implements the OpenMP fork-join model for parallel regions by separating the GPU threads used to launch the kernel into a master thread and a set of worker threads. The master thread is used to execute any sequential code that may be present in the target region, and it is also responsible for setting up the execution of any parallel region contained in the target region. While the master thread executes, the worker threads sit idle at a barrier. When the master thread completes the setup, it reaches a barrier and thus lets the worker threads execute the outlined code corresponding to a parallel region.

This code generation approach illustrated by Figure 2a is necessary in order to handle generic target regions. However orchestrating CUDA threads to fit the OpenMP fork-join programming model introduces significant runtime overhead for target regions that contain no sequential code.

We observe that this generic scheme is not necessary for target regions that contain no sequential user code. That is, when the compiler can prove that all threads in a team execute the same parallel loop, the code generated can be simplified. All threads in the team can be directly used to execute the parallel loop iterations, and the runtime calls that were required to setup the execution of a generic parallel region can be elided (Figure 2b). We label this specialized code generation scheme "SPMD" because all threads execute the same code.

Figure 2a: generic pseudo-code for a target region

```
void gpu_offloaded_kernel() {
  if (warpId == masterWarpId) {
    __kmpc_for_static_init(lb, ub, st);
    i = lb;
    do {
      …
      __kmpc_kernel_prepare_parallel(…
            &__xl_outlined_parallel_region);
      llvm.nvvm.barrier(); //launch barrier
      llvm.nvvm.barrier(); //completion barrier
      i = i + st;
    } while (i < ub);
    __kmpc_kernel_deinit();
  } else {
    do {
      llvm.nvvm.barrier(); //launch barrier

      // Initialize function pointer workFnPtr
      // with the address of the outlined
      // parallel region
      // (__xl_outlined_parallel_region)
      //
      if (__kmpc_kernel_parallel(workFnPtr))
        // Call through function pointer
        (*workFnPtr)();
      llvm.nvvm.barrier(); //completion barrier
    } while (!done);
  }
}
```

The distribute parallel loop in Figure 1 satisfies all the requirement necessary to apply the SPMD code generation scheme, given that all teams in the region will execute the same parallel loop and no user code is present outside the loop. Figure 2b illustrate the SPMD code generation scheme corresponding to that example. This pseudo-code is similar to the code associated with a straightforward CUDA implementation for the same kernel.

Figure 2b: SPMD pseudo-code for a target region

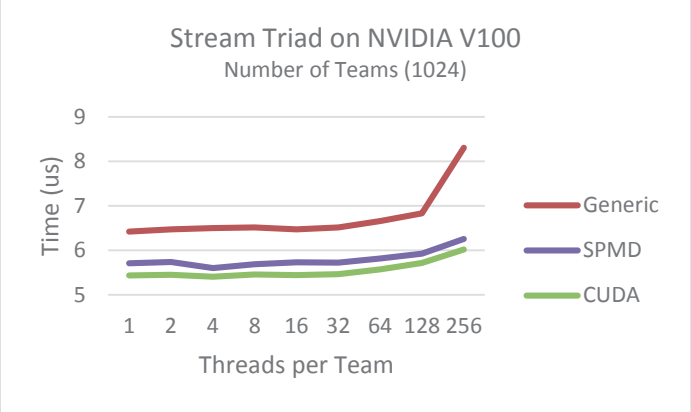
```
void gpu_offloaded_kernel() {
  __kmpc_spmd_kernel_init(blockDimx);
  i = blockDimx * blockIdx + threadIdx;
  bump = gridDimx * blockDimx;
  do {
    a(i) = b(i) + scalar * c(i)
    i = i + bump;
  } while (i < n);
```

```
}
```

To evaluate the runtime overhead associated with the generic and the SPMD code generation schemes we run the OpenMP Stream Triad microkernel and an equivalent CUDA C version [Stream]. For all versions, we run the micro benchmark 10 times and obtained the average runtime. In order to evaluate the effect of grid selection size used to run the microkernel on the code generation overhead we have varied the number of threads per team from 1 to 256 while fixing the number of teams to 1024. The arrays sizes used in the Stream benchmark have been selected to be equal to the number of teams multiplied by the size of a team.

The results in Figure 3 illustrate that the generic code generation scheme suffers an overhead, compared to the CUDA C version, ranging from 15% to 28%. Using the SPMD code generation scheme the runtime overhead, compared to the CUDA C version, ranges from 3% to 5%, showing that the SPMD code generation scheme leads to runtime performance that is quite comparable to that of the CUDA implementation. The additional overhead presented by the SPMD code generation scheme can be attributed to the check the compiler generates to handle cases where the number of array elements is not dividable by the grid size (not shown in Figure 2b).

Figure 3: performance comparison for the Stream Triad microkernel



As mentioned before, not all target regions can be generated in SPMD mode. The SPMD code generation scheme can be used when the code in the dynamic extent of the target region can be safely executed by all the threads. The XL compiler attempts to analyze the target region to identify program statements and OpenMP directives that may prevent SPMD code generation. In particular it analyzes the context of function calls within the scope of the target region based on their location with respect to parallel regions.

There are two situations to consider depending on the location of the call within the target region. As a first case consider the code in Figure 4. Here the call to the function foo() is within the scope of the *distribute parallel do* construct.

Figure 4: target region containing a call inside a parallel region

```
!$omp target teams distribute parallel do
      map(from: a) map(to: b,c,scalar)
do j = 1,n
  call foo(a, b, c, scalar)
end do
```

In this case the SPMD code generation scheme can be safely used even if the body of foo() contains a parallel region. This is because the OpenMP specification permits nested parallel regions to be run sequentially, and therefore it is not necessary to spawn new threads to handle the nested parallelism that may be present in the body of the function called. The entire target region is executed by all the threads the OpenMP runtime system used to invoke it.

As a second example consider the code in Figure 5a. Here the call to the function foo() is outside of the scope of the *distribute parallel do* construct. In this case the compiler needs to conservatively assume that the body of the function called might contain an SPMD preventing statement or construct. An example of an SPMD preventing condition is a statement that is executed conditionally based on the thread ID and that therefore needs to be executed by a specific thread in a team of threads.

Figure 5a: target region containing a call outside of a parallel region

```
!$omp target teams
      map(from: a) map(to: b,c,scalar)
!$omp distribute parallel do
do j = 1,n
  a(j) = b(j) + scalar * c(j)
end do
call foo(a, b, c, scalar)
!$omp end target teams
```

In order to determine whether the SPMD code generation scheme can be used, the compiler needs to perform a context sensitive inter-procedural analysis. That is, it needs to analyze the characteristic and location of the calls through the call graph.

The inter-procedural analysis is also useful to discover opportunities for SPMD code generation that a simplistic intraprocedural analysis would miss. The code in Figure 5b can be used to illustrate this latest point. Here the *target team* region contains a function call to foo() which has been declared as a *declare target* subroutine. The body of the declare target subroutine contains a *distribute parallel do*. In the absence of inter-procedural support, the SPMD analysis would fail to detect that the loop in foo() satisfies the requirement for SPMD code specialization and default to the generic code generation scheme. This is a simple example to illustrate the concept, and similar code pattern is common in practice due to use of *declare target* functions.

Figure 5b: function with distribute parallel loop called within a target region

```
!$omp target teams
     map(from: a) map(to: b,c,scalar)
call foo(a, b, c, scalar)
!$omp end target teams

subroutine foo(a, b, c, scalar)
  !$omp declare target
  !$omp distribute parallel do
  do j = 1,n
    a(j) = b(j) + scalar * c(j)
  end do
end subroutine foo
```

Because the function candidate for code specialization might have multiple callers in different contexts (possibly in other compilation units), the SPMD code specialization needs to clone the body of the function and generate a generic version to account for unknown calling contexts. All SPMD calling contexts in the same translation unit may though use the same specialized version of foo().

As an example, consider the code in Figure 6a. The function 'funcN' containing a *distribute parallel do* can be invoked either from a call chain originating within an OpenMP target region or via a call chain originating from a generic context (e.g. not inside an OpenMP construct). Given that 'funcN' can be executed through different calling contexts, both SPMD and generic versions need to be generated for it, and part of the call chain leading to it has to be cloned. The resulting call graph can be found in Figure 6b. We note that it is unnecessary to create a specialized version for 'func1' because the target team region is the only caller, or more specifically, the target team region dominates it. Similarly, it is not necessary to clone 'funcM' since it does not reach the function containing the SPMD specialization opportunity. In general, a specialized version needs to be created for all functions that can reach the function containing the *distribute parallel do* construct and may be reached from the target team region but are not dominated by it. Later phases in the compiler can inline the cloned functions in their callers, thus reducing function call overhead.

In conclusion, when proven safe, SPMD code generation can provide a large performance gain compared to a more general code generation scheme for target team regions running on a GPU, yielding performance similar to CUDA code (Figure 3). Also, by implementing the SPMD code generation analysis interprocedurally more opportunities for this transformation scheme can be discovered by the compiler (Figure 6a and Figure 6b).

2 Device Function Devirtualization

The XL compiler leverages a similar generic code generation scheme to the clang compiler for efficient coordination of threads in OpenMP target regions as described in [Bertolli05]. In cases where SPMD code specialization is not feasible, the typical pattern of code generated by the compiler is illustrated in Figure 2a. It is important to observe that an indirect call through a function pointer is used to invoke the procedure the compiler generate to encode parallel regions present in the target region. This indirect function call results in a number of performance issues.

Figure 6a: call graph before SPMD specialization

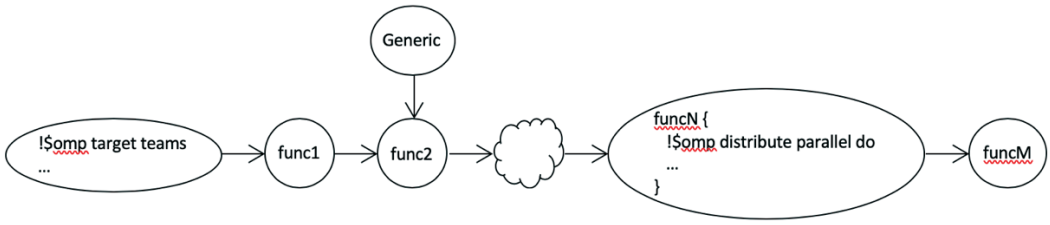
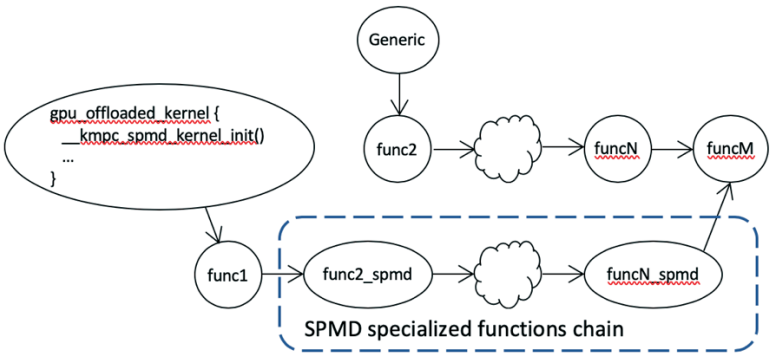


Figure 6b: call graph after SPMD specialization



One obvious drawback of having indirect function calls is that inlining opportunities are prevented. Another problem is that it causes register pressure to increase by a large amount on NVIDIA GPUs[1], potentially resulting in reduced kernel occupancy and underutilization of the GPU processing resources.

In order to avoid these performance costs, the XL compiler uses information available in the program call graph to replace the indirect call with a sequence of compare and branch instructions followed by a direct function call. In general, there is one set of compare and branch instruction per possible target of a worker function pointer. However, in the case where the target region contains a single parallel region the extra overhead of compare and branch instructions are avoided by control-flow simplification passes that run after this optimization resulting in the generation of a single direct function call. This leads to better register utilization by the produced kernel. Figure 7 illustrates the output produced after the devirtualization and control-flow optimizations for a target region containing a single parallel region.

Figure 7: parallel region call devirtualization

```
void gpu_offloaded_kernel(...) {
  if (threadIdx != MASTER)
    goto worker_label;
  ...
  __kmpc_kernel_prepare_parallel(
              ...&__xl_outlined_parallel_region);
  llvm.nvvm.barrier0();
  llvm.nvvm.barrier0();

worker_label:
 do {
   llvm.nvvm.barrier0();
   // Initialize function pointer workFnPtr0 with
   // the address of the outlined parallel region
   // (__xl_outlined_parallel_region).
   $_$execStatus0 =
         __kmpc_kernel_parallel(&workFnPtr0,0);
   ...
   // *** Direct Function Call *** //
   __xl_outlined_parallel_region();
   llvm.nvvm.barrier0();
 } while (!done);
}
```

Figure 8a shows the impact of this optimization on register pressure for a matrix multiply implementation targeting NVIDIA V100 GPU. Figure 8b shows the impact of the reduction in register pressure on the actual performance of that same application. The 42% improvement in performance is the result of increased occupancy which was inhibited by the larger number of registers in the non-optimized case.

3 Data Mapping Optimizations

3.1 Firstprivate Optimization

The overhead and complexity of managing data transfers between host and device has always been one of the challenges associated with offloading computational kernels to an accelerator.

Figure 8a: Impact of parallel region calls devirtualization on register pressure

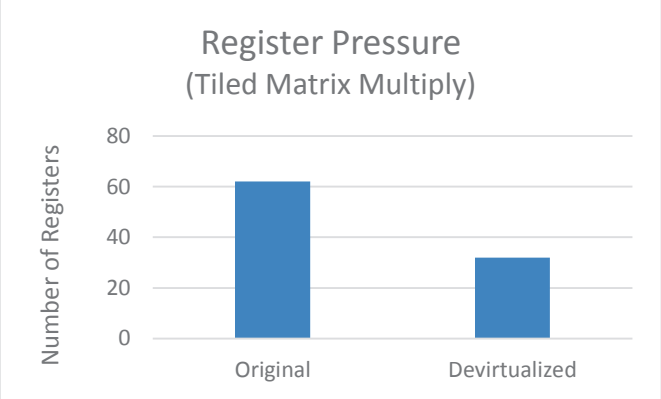
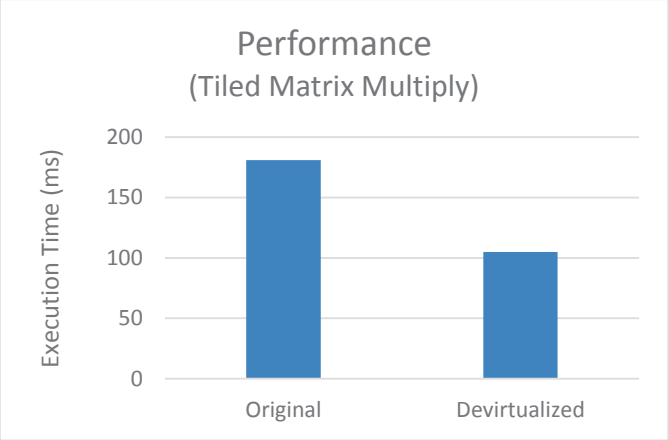


Figure 8b: Impact of parallel region calls devirtualization on performance



The OpenMP data mapping directives allow the programmer to be explicit about how and when data is going to be transferred between host and device, but it also defines a number of rules governing how certain data objects are implicitly mapped. For instance, in the absence of *defaultmap* and explicit data-sharing and data-mapping attributes, simple Fortran scalar variables (those that do not have the TARGET, ALLOCATABLE, or POINTER attribute) implicitly take on the *firstprivate* data-sharing attribute. These types of scalars are fairly common in scientific programs, and their values need to be communicated from host to device in order for the offloaded regions to perform their computations.

[1] The impact on register pressure has not been examined on other accelerators, however this optimization is expected to benefit other platforms as well due to inlining and reduction in call overhead.

```
==84361== NVPROF is profiling process 84361, command: ./a.out
==84361== Profiling application: ./a.out
==84361== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   52.85%  4.2877ms        62  69.156us  2.2720us  74.944us  [CUDA memcpy DtoH]
                   47.04%  3.8161ms       168  22.715us  1.4400us  36.512us  [CUDA memcpy HtoD]
                    0.11%  9.0240us         1  9.0240us  9.0240us  9.0240us  __xl___m_NMOD_hpc_kernel_l32_OL_1
```

Figure 10: nvprof trace with data mapping optimizations disabled

Furthermore, compiler-generated scalar temporaries that are defined outside of a particular target region and then used inside that region are extremely common. The firstprivate implicit data sharing attribute applies to these temporaries as well.

One way to make the value of a firstprivate variable available to the device is by issuing a data transfer from host to device. However, such data transfers come with significant performance cost that cannot always be circumvented with available techniques such as overlapping of data transfers with execution of available kernels whose data dependencies are satisfied. In order to address this performance problem, the XL compiler performs an optimization whereby certain firstprivate variables are passed to the kernel as by-value arguments.

This is achieved by careful examination of the list of firstprivate variables in a target region (including implicitly defined ones), looking for candidates whose data type sizes are small enough to fit in a device register. Consideration is also made towards the total number of parameters being passed to the kernel, as there are limits to the total size of the parameter area depending on the target device architecture.

3.2 Unused Map Removal

The OpenMP specification [OpenMP] allows explicit mapping of variables that are not used in a target region. Such data maps are useless, incurring unnecessary overhead, and may occur in real programs due to programming oversights or over-pessimistic assumptions about aliasing rules. It is also possible for a certain variable to appear in a target region in an area of code that is never executed or be used in a statement that can be removed as a result of data flow optimizations. The same situation may happen for compiler-generated code where some temporaries may no longer be needed after dead code elimination. The XL compiler is well equipped to identify unused data maps through its sophisticated alias analysis and eliminate unnecessary data transfers between host and device.

3.3 Optimization Example and Performance Impact

The impact of the data mapping optimizations we described in the previous sections is significant for many target regions (i.e. GPU kernels). Figure 9 illustrates an example exhibiting the data-transfer optimization opportunities mentioned above. In this example, the XL compiler is able to eliminate the data transfer of the `unused` array to and from the device. It's also able to pass the firstprivate variable `fp` by-value thereby eliminating another data transfer from host-to-device. A total of 4100 bytes of host-to-device and 4KB of device-to-host data transfers are optimized away in this case.

Figure 9: target region with unused mapped variables

```
module omp_example
  implicit none
    contains

  subroutine sub1(input)
    real, intent(in) :: input
    real :: unused(1024), fp, res
    res = 0.0
    unused = 0.0
    fp = input*input

    !$omp target map(tofrom: res, unused)
                firstprivate(fp)
      res = res+fp
    !$omp end target

    print *, "result = ", res
    if (unused(1) .ne. 0.0 .or.
        unused(1024) .ne. 0.0) then
      error stop
    end if
  end subroutine sub1
end module omp_example

program main
  use omp_example
  implicit none
  call sub1(3.0)
end program main
```

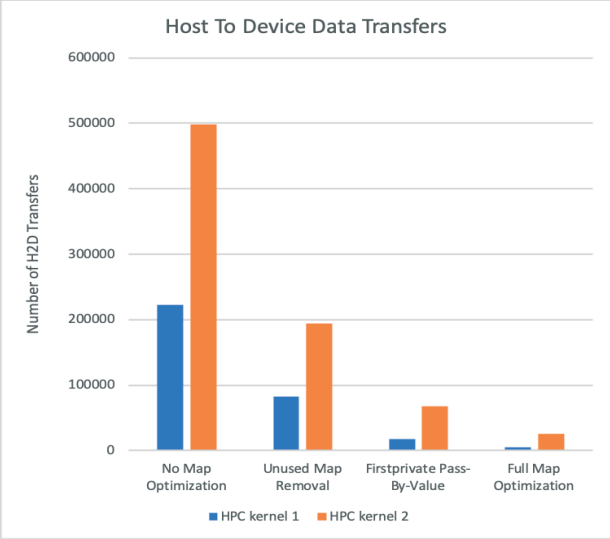
Profiling traces are also shown for a proxy application containing typical code pattern found in HPC Fortran applications with adjustable arrays and a number of scalar variables. Figure 10 shows the traces obtained from 'nvprof --print-gpu-summary' when the data mapping optimizations are disabled, while Figure 11 shows the number of host-to-device data transfers when the optimization is enabled. As shown in the profile summary the XL compiler is able to reduce the number of host-to-device data transfers from 168 to 125, a reduction of about 25%.

Similarly, the number of data transfers with and without the optimizations described in section 3.1 and 3.2 are compared for a representative HPC kernel. Figure 12 shows the impact of each individual optimization as well as the combined effect on the total number of host-to-device data transfers.

```
==84212== NVPROF is profiling process 84212, command: ./a.out
==84212== Profiling application: ./a.out
==84212== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   54.09%  4.4061ms        62  71.066us  1.9200us  76.544us  [CUDA memcpy DtoH]
                   45.81%  3.7320ms       125  29.855us  1.6000us  39.456us  [CUDA memcpy HtoD]
                    0.10%  8.3840us         1  8.3840us  8.3840us  8.3840us  __xl___m_NMOD_hpc_kernel_l32_OL_1
```

Figure 11: nvprof trace with data mapping optimizations enabled

Figure 12: Performance impact of map optimizations



4 GPU Grid Geometry Selection

The OpenMP specification [OpenMP] allows users to specify target regions that will be executed on accelerator devices such as GPUs. The code section of a target region is outlined by the compiler as a standalone function called a kernel. The kernel is then executed by threads on the device. NVIDIA Tesla V100-SXM2 GPU used in IBM AC922 is composed of 80 Streaming Multiprocessors (SMs), each SM has 64 cores capable of running 2 Warps of 32 threads each simultaneously [Nvidia]. Threads are grouped into blocks and all threads in the same block execute on a single SM. The number of thread blocks and the number of threads per thread block must be specified when a kernel is launched on the GPU. This is referred to as a grid geometry. For optimal performance, proper selection of the grid geometry for a given kernel is critical to efficiently exploit the target GPU hardware resources. Because different target regions have different characteristics, it is challenging for the compiler runtime to determine the best GPU grid geometry for each and every target region. To further complicate the matter, necessary information may not be available for the compiler runtime to formulate a suitable grid geometry for a particular target region.

T. Lloyd, A. Chikin, S. Kedia et al [Lloyd18] built a machine learning model to research into GPU grid geometry selection and characteristics of a kernel that limits potential occupancy or GPU utilization, such as the loop trip count of a kernel, the number of registers and amount of shared memory used by a kernel. Based on this research, they categorized kernels into three classes and proposed heuristics for the compiler runtime to determine the GPU grid geometry: 1) for kernels whose loop trip count is less than or equal to the number of available SMs, use the loop trip count as the number of thread blocks and 1 thread for each thread block; 2) for kernels whose loop trip count are larger than the number of available SMs, the number of thread blocks is determined based on the loop trip count, threads per block, the registers and shared memory used by the kernel; 3) If the loop trip count exceeds the product of threads per block and number of blocks calculated from the resource usage of the kernel, the heuristics uses 96 threads per block and sets the number of blocks to be the number of blocks from the resource calculation.

The XL compiler has adopted the heuristics proposed by T. Lloyd, A. Chikin, S. Kedia et al and enhanced it to address issues encountered in practice. For example, the heuristics relies on the loop trip count for determining the thread block size. However, the loop trip count is not always available. Figure 13 is an example where the compiler is unable to determine the loop trip count of the distribute parallel loop directly contained by the target team region. Depending on the value of variable *j*, the loop trip count can be either 100 or 200.

In case the loop trip count is not available, the XL compiler runtime attempts to select a grid size sufficiently large as to yield 100% GPU occupancy when the kernel runs. The number of thread blocks is chosen to be a multiple of the number of SMs on the target GPU using the following generalized formula.

$$NumOfBlocks = \frac{NumOfSMs \times NumOfWarpsPerSM \times ThreadsPerWarp}{ThreadsPerBlock}$$

The heuristics proposed in [Lloyd18] uses 96 threads per block and caps the number of blocks to be the number of blocks from the resource calculation even if the loop trip count exceeds the product of threads per block and number of blocks from the calculation. This reduces the parallelism because many loop iterations are not covered by the grid size and have to be queued. The XL compiler addresses such scenario by increasing the number of threads per block instead of using a fixed number

of threads per block to achieve better parallelism and performance.

Figure 13: Target team region containing distribute parallel loops

```
!$omp target teams
  if (j .eq. 1) then
    !$omp distribute parallel do
    do i = 1, 100
      c(i) = a(i) + b(i)
    enddo
    !$omp end distribute parallel do
  else
    !$omp distribute parallel do
    do i = 1, 200
      c(i) = a(i) + b(i)
    enddo
    !$omp end distribute parallel do
  endif
!$omp end target teams
```

The enhanced heuristics used by the XL compiler runtime to select the kernel grid size yield an average 2x speedup on the SPEC ACCEL V1.2 suite as compared with the original implementation [Table 1].

Table 1: Speedup of the tuned kernel grid selection heuristics on the SPEC ACCEL V1.2 benchmark suite

Benchmark	**Original Heuristics**	**Enhanced Heuristics**	**Speedup**
	Time(s)	Time(s)	
503.postencil	13.31	12.61	1.06
504.polbm	22.62	22.61	1.00
514.pomriq	33.54	37.47	0.90
550.pmd	168.15	23.32	7.21
551.ppalm	9514.99	1527.32	6.23
552.pep	177.32	177.88	1.00
553.pclvrleaf	2202.19	567.28	3.88
554.pcg	103.65	90.87	1.14
555.pseismic	38.54	39.49	0.98
556.psp	33.84	33.84	1.00
557.pcsp	99.87	94.62	1.06
559.pmniGhost	78.42	48.36	1.62
560.pilbdc	34.07	33.92	1.00
563.pswim	40.85	42.26	0.97
570.pbt	122.95	80.48	1.53
Average speedup			2.04

The performance experiment was carried out using a Power9 system with NVIDIA Tesla V100-SXM2 GPUs. We note that while most benchmarks benefit from our enhanced grid selection heuristic, some show large speedups (e.g. 550.pmd).

While the XL compiler is able to determine the loop trip count for most target regions containing a distribute parallel loop, in benchmark 550.pmd the loop trip count is undeterminable at compilation time and therefore unknown at the time the kernel is launched. Therefore our enhanced heuristics uses 128 threads per block to run the kernel, a value suitable for most cases from our tuning experiments and calculates the number of thread blocks from the formula above.

Conversely the original grid selection heuristic took the maximum number of threads per block allowed by the target GPU for the kernel (this is obtained from CUDA runtime routine cuFuncGetAttribute(), in this case 640) and the default value of 128 as the number of the blocks (due to the unavailability at runtime of the loop trip count). This was insufficient to achieve good occupancy, and as a result of better occupancy the enhanced heuristics yields a 7x speedup for this benchmark.

5 XL Fortran diagnostics of mapping errors

The *target* directive's *map* clause is the primary method provided by the OpenMP specification [OpenMP] for allocating and transferring data from host to target device and vice versa. To avoid unnecessary data transfers, it is common practice to leverage the OpenMP *target enter/exit data* or *target data* directives. These directives can be used to reduce the transfer cost when the same offloaded data needs to be processed by multiple OpenMP target regions. In addition, this strategy can improve overall performance when data-offloading and target region execution are carried asynchronously with respect to each other.

Although pre-mapping data is beneficial to reduce data transfers between host and device, it is also prone to errors in practice. This difficulty arises because the user needs to follow a fairly complex set of semantic constraints that affect the map clause. For example, as required by the OpenMP specification, a map clause is not allowed to extend the device storage that have been allocated by a prior map for the same object. Such runtime error can occur when an application pre-maps sections of an array and subsequently launches a target region that requests offloading of a larger overlapping section or the whole array. We note that the later map operation may be implicitly inserted by the compiler in accordance to the OpenMP default mapping rules. Also, in Fortran, offloading of derived-type components often results in similar mapping errors as well.

As a motivating example consider the snippet of Fortran code in Figure 14. On line 16 a section of the `*data_matrix*` array is mapped to the target device data environment via the use of *target enter data* directives. Subsequently at lines 37-39 a target region (kernel-a) references the `*data_matrix*` array and maps the whole array explicitly. This yields to a violation of the mapping semantics and ultimately a failure when the program is run. Another example is provided by kernel-b in Figure 14. In this case, components of a Fortran derived type object `*g_compute_env*` is mapped at lines 16 and 23. Subsequently at lines 41-44 a target region (kernel-b) references a field in `*g_compute_env*` and this causes the compiler to implicitly map the entire aggregate, leading to a runtime error.

Figure 14: (kernel-a) Pre-mapping of an array section is followed by mapping of the whole array. (kernel-b) Pre-mapping of a structure section is followed by implicit mapping of the whole structure.

```
 2 type dt
 3   real :: config1, config2
 4   real, pointer :: data_matrix(:, :)
 5   real :: config3, config4
 6 end type dt
 8 type(dt) :: g_compute_env
 ...
16 !$OMP TARGET ENTER DATA MAP(TO:
g_compute_env%data_matrix(:, slice))
 ...
23 !$OMP TARGET ENTER DATA MAP(TO:
g_compute_env%config2)
 ...
! kernel-a
37 !$OMP TARGET MAP(g_compute_env%data_matrix)
38   g_compute_env%data_matrix = 3.14
39 !$OMP END TARGET
 ...
! kernel-b
41  !$OMP TARGET
42   g_compute_env%config4 = 10
43   g_compute_env%config2 = 2
44  !$OMP END TARGET
```

During the early development stage of the CORAL project, several application teams frequently encountered these types of errors. The XL OpenMP runtime was able to report semantic violations related to a mapping mistake but, while the diagnostic information provided was useful to identify the kind of error that was causing the application to fail, tracing the error back to the source program (i.e. to the particular map clauses at fault) proved to be non-trivial at best and infeasible in general (given the size and complexity of the typical HPC application).

In order to help identifying the conflicting map clauses, the XL Fortran compiler developed a mechanism to relate map clauses to their corresponding source location. When the application is built with the -qinfo=omperrtrace option, the error reported by the OpenMP runtime contains detailed source level information. For each of the conflicting map operations, the information includes source file name, line number, and column number. In addition, the report includes the host-address range and length of the mapped object, whether the map is for a structure section, and whether it is user-specified or compiler-generated. Figure 15 shows the error output generated by the code snippet in Figure 14.

Conclusion

This paper describes aspects of the OpenMP 4.5 implementation in the IBM XL C/C++ and XL Fortran OpenMP production compilers. We introduce an inter-procedural context sensitive static analysis used to specialize code generation for target regions offloaded for execution on a NVIDIA GPU and evaluate its impact on a Stream Triad microbenchmark. We then show a function devirtualization technique that enables inlining and helps reduce register pressure. We also show a set of optimizations used to reduce the overhead of data transfers between the host system and the GPU device followed by an evaluation of the heuristic used by the XL OpenMP runtime system to select grid sizes used to launch offloaded target regions. We evaluate the impact of the selection algorithm using the SPEC ACCEL V1.2 benchmarks [SpecAcc], results achieved are very encouraging and yield an average 2X improvement. Finally, we describe a diagnostic feature implemented in the XL Fortran compiler to aid in debugging OpenMP applications offloaded to accelerators.

Figure 15: Error output with *-qinfo=omperr*trace for the example program in Figure 14

```
  // Error from kernel-a
  Current map  [0x1001056b9b0, 0x1001056bb3f] len=400 :
     Src: file1.f Line: 37 Column: 25 Listitem: g_compute_env%data_matrix
  Previous map [0x1001056b9b0, 0x1001056b9d7] len=40 :
     Src: file1.f Line: 16 Column: 39 Listitem: g_compute_env%data_matrix
  1587-171 Error when processing an explicit map that fully includes the memory of a single prior map on the target device 0.  Prior
map starts at host address 0x1001056b9b0.  The program will stop.

  // Error from kernel-b
  Current map     [0x1039fe30, 0x1039fe8f] len=96 :
     Src: file1.f Line: 42 *Implicit-Map*  Listitem: g_compute_env
  Previous map(1) [0x1039fe34, 0x1039fe37] len=4 :
     Src: file1.f Line: 23 Base symbol: g_compute_env | section:  [ g_compute_env%config2 , g_compute_env%config2]
     Src: file1.f Line: 23 Column: 39 Listitem: g_compute_env%config2
  Previous map(2) [0x1039fe38, 0x1039fe87] len=80 :
     Src: file1.f Line: 16 Base symbol: g_compute_env | section:  [ g_compute_env%data_matrix(pointer) ,
g_compute_env%data_matrix(pointer)]
1587-167 Encountered an ambiguous map that fully includes the memory of several prior maps on the target device 0, including
prior maps that start at host addresses 0x1039fe34 and  0x1039fe38.  The program will stop.
```

Acknowledgment

We would like to sincerely thank Alexandre Eichenberger, Arpith Jacob, and Kevin O'Brien for sharing their expertise on the OpenMP runtime system. We are grateful to Leopold Grinberg for sharing his experience with OpenMP and its use in the CORAL proxy applications. We would also like to thank the entire IBM XL compiler team for the collaboration that enabled the design and implementation of the ideas described in this paper. We also acknowledge the support by the CORAL project, partially funded U.S. Department of Energy contract no B604142.

References

[AC922] IBM Power System AC922 Technical Overview and Introduction, [Online]. Available: http://www.redbooks.ibm.com/abstracts/redp5494.html (URL)

[CUDAC] CUDA C Programming Guide, [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (URL)

[CUDAF] CUDA Fortran Programming Guide, [Online]. Available: https://www.pgroup.com/resources/docs/19.7/pdf/pgi19cudaforug.pdf (URL)

[OpenCL] OpenCL 2.2 API Specification, [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_API.html (URL)

[OpenMP] OpenMP Application Programing Interface, Nov 2015, Version 4.5 [Online]. Available: http://openmp.org (URL)

[IBMXL] IBM XL C/C++ and XL Fortran compilers on Power architectures overview, [Online]. Available: https://www.ibm.com/support/pages/ibm-xl-cc-and-xl-fortran-compilers-power-architectures-overview (URL)

[Stream] STREAM Benchmark in CUDA C++, [Online]. Available: https://github.com/bcumming/cuda-stream (URL)

[Bertolli14] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O'Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave. Coordinating gpu threads for openmp 4.0 in llvm. In Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM-HPC'14, pages 12–21, Piscataway, NJ, USA, 2014. IEEE Press.

[Nvidia] NVIDIA Corporation, NVIDIA TESLA V100 GPU ARCHITECTURE: The World's Most Advanced Data Center GPU, [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (URL)

[Lloyd18] T. Lloyd, A. Chikin, S. Kedia et al, "Automated GPU Grid Geometry Selection for OpenMP Kernels," Proc. 2018 30th International Symposium on Computer Architecture and High-Performance Computing, pp. 442-449, 2018 (conference proceedings)

[SpecAcc] SPEC ACCEL benchmark suite V1.2, [Online]. Available: https://www.spec.org/accel/ (URL)

Ettore Tiotto *IBM Canada Laboratory, Compiler Development, 8200 Warden Ave, Markham, ON L6G 1C7, Canada (etiotto@ca.ibm.com).* Mr. Tiotto is a Senior Compiler Developer at the IBM Toronto Laboratory. He graduated "Summa Cum Laude" in Physics from the University of Torino, Italy, in 1996. Since joining the IBM XL compiler Team in 1999 he has participated in the development of numerous releases of the industry leading XL C/C++ and Fortran compilers for POWER. Since 2015 he works as a technical architect in the XL compiler development team on the CORAL project for which he received an IBM Outstanding Technical Achievement Award. He is an author or coauthor of more than 31 patents and 14 technical papers. Interests include skiing, running, and cooking.

Bardia Mahjour *IBM Canada Laboratory, Compiler Development, 8200 Warden Ave, Markham, ON L6G 1C7, Canada (bmahjour@ca.ibm.com).* Mr. Mahjour is a Senior Compiler Developer at the IBM Toronto Laboratory. He earned a BSc. degree in Software Engineering with distinction from the University of Calgary in 2007. Since joining the IBM XL Compiler Team, he has worked on numerous releases of the industry leading XL C/C++ and Fortran compilers. Since 2015 he contributed to the CORAL project in various leadership capacities for which he received an IBM Outstanding Technical Achievement Award.

Whitney Tsang *IBM Canada Laboratory, Compiler Development, 8200 Warden Ave, Markham, ON L6G 1C7, Canada (whitneyt@ca.ibm.com).* Miss Tsang is an Advisory Compiler Developer at the IBM Toronto Laboratory. She graduated "With Distinction - Dean's Honours List" in Computer Science from the University of Waterloo, Canada, in 2016. Since joining the IBM XL compiler Team, she has worked on the industry leading XL C/C++ and Fortran compilers for POWER. Since 2017 she has worked in the XL compiler development team on the CORAL project.

Xing Xue *IBM Canada Laboratory, Compiler Development, 8200 Warden Ave, Markham, ON L6G 1C7, Canada* (xingxue@ca.ibm.com). Mr. Xue is a compiler developer at IBM. He received his PhD in computer science from Nanjing University, China, in 1988. Since joining the IBM XL compiler Team, he has worked on numerous components of XL C/C++ and Fortran compilers for POWER and IBM Z, including the OpenMP runtime for the CORAL project. His interest lies in the area of distributed and shared memory parallel systems and compilers.

Tarique Islam *IBM Canada Laboratory, Compiler Development, 8200 Warden Ave, Markham, ON L6G 1C7, Canada (*tislam@ca.ibm.com*)*. Mr. Islam is a compiler developer at IBM. He received his PhD in computer science from the University of Waterloo, Canada, in 2007. As a member of the XL compiler team on POWER, he has worked on many projects including OpenMP and CUDA-Fortran. His interest lies in the area of compiler-optimization, graph theory, and computational complexity.

Wang Chen *IBM Canada Laboratory, Compiler Development, 8200 Warden Ave, Markham, ON L6G 1C7, Canada (wdchen@ca.ibm.com).* Mr. Chen is a senior manager for compiler development at IBM. He has extensive experience in

compiler optimization for various workloads, including commercial, analytics, and High-Performance Computing. He has led the IBM XL compiler team in development of C, C++, and Fortran OpenMP compilers to exploit NVIDIA GPU in the OpenPOWER systems, which enabled order of magnitude speed up for key US Department of Energy CORAL workloads. Wang is also a Research Collaboration Manager for Centre for Advanced Studies and was a co-chair of the program committee for CASCON 2018. His interest includes programming models for accelerated computing and cognitive based compiler optimization.