Parallel Kernel

Related terms:

Parallel Machine, Compiler, Programming Model, Sequential Machine

View all Topics

Parallel Programming with OpenACC

Programming Massively Parallel Processors (Second Edition), 2013

Asynchronous Computation and Data Transfer

OpenACC provides support for asynchronous computation and data transfer. An async clause can be added to parallel, kernels, or update directive to enable asynchronous execution. If there is no async clause, the host process will wait until the parallel region, kernels region, or updates are complete before continuing. If there is an async clause, the host process will continue with the code following the directive when the parallel region, kernels region, or updates are processed asynchronously. An asynchronous event can be waited by using the wait directive or OpenACC runtime library routines.

In the Jacobi relaxation example in Figure 15.20, the update of the host copy of field (line 37) and the display of it on the host could happen in parallel with the compute of tmpfield (lines 31 and 32) on the device.

In Figure 15.21, to enable the asynchronous execution, we move the update and display in between the two parallel regions, add an async clause to the parallel directive at line 31, and add a wait directive before the second parallel directive at line 33.

```
26 void transform() {
27      ...
28      #pragma acc data copy(field) create(tmpfield)
29      {
30          for (int i=0; i<128; i++) {
31              #pragma acc parallel async
32              relaxation(field, tmpfield);
35              if (check) {
36                 #pragma acc update host(field)
37                 display(field);
38              }
33-1            #pragma acc wait
33-2            #pragma acc parallel
34              relaxation(tmpfield, field);
39          }
40      }
41      ...
42  }
```

Figure 15.21. async and wait.

We can replace the wait directive with a call to the OpenACC acc_async_wait_all() routine and achieve the same effect. OpenACC provides a richer set of routines to support the asynchronous wait functionality, including the capability to test whether an asynchronous activity has completed rather than just waiting for its completion.

> Read full chapter

The Essential OpenACC

Thomas Sterling, ... Maciej Brodowicz, in High Performance Computing, 2018

16.5.2 Kernels Construct

The compiler encountering the **kernels** directive performs the analysis of marked sections of the code and converts these into a sequence of parallel kernels that will be executed in order on the accelerator device. The number of gangs and workers and vector size may be different for each such kernel. The workload subdivision is typically performed in a way that creates one kernel for each loop nest present in the code. The primary difference between the **kernels** construct and the **parallel** directive is that the latter relies on the programmer to configure various parameters that divide the workload across accelerated execution resources. Thus the use of the **kernels** directive is recommended for beginners to OpenACC programming, but it may not always yield the best-performing code. Its syntax is shown below:

#pragma acc kernels *[clause-list]*

structured-block

The **kernels** construct accepts **async** and **wait** clauses that behave as described for the **parallel** clause, as well as data management clauses (discussed further in Section

16.5.3). Similar restrictions to those of the **parallel** directive apply: the code may not branch out or into the accelerated region.

Example:

```
1  #include <stdio.h>
2
3  const int N = 500;
4
5  int main() {
6     // initialize triangular matrix
7     double m[N][N];
8     for (int i = 0; i < N; i++)
9        for (int j = 0; j < N; j++)
10          m[i][j] = (i > j)? 0: 1.0;
11
12    // initialize input vector to all ones
13    double v[N];
14    for (int i = 0; i < N; i++) v[i] = 1.0;
15
16    // initialize result vector
17    double b[N];
18    for (int i = 0; i < N; i++) b[i] = 0;
19
20    // multiply in parallel
21    #pragma acc kernels
22    for (int i = 0; i < N; i++)
23       for (int j = 0; j < N; j++)
24          b[i] += m[i][j]□v[j];
25
26    // verify result
```

```
27      double r = 0;
28      for (int i = 0; i < N; i++) r += b[i];
29      printf("Result: %f (expected %f)\n", r, (N+1)□N/2.0);
30  }
```

Code 16.3. Accelerated matrix–vector multiply using the kernels directive.

The program listed in Code 16.3 performs multiplication of a matrix and a vector, the dimensions of which are known at compile time and fixed. The accelerated region of code follows the **kernels** directive in line 22 and contains a loop nest: the outer loop iterates over matrix rows (index i) and the inner loop over the columns (index j). Unlike Code 16.2, the execution of the parallel region is synchronous (there is no **async** clause), meaning that the program will not proceed to result verification until the accelerated kernel computation is finished. The result of program execution is shown below:

Result: 125250.000000 (expected 125250.000000)

> Read full chapter

Parallel programming with OpenACC

Jeff Larkin, in Programming Massively Parallel Processors (Third Edition), 2017

Now that we have learned to design and express parallel algorithms in CUDA C, we are in a strong position to understand and use parallel programming interfaces to rely on the compiler to do the detailed work. OpenACC is a specification of compiler directives and API routines for writing data parallel code in C, C++, or Fortran that can be compiled to parallel architectures, such as GPUs or multicore CPUs. Rather than requiring the programmer to explicitly decompose the computation into parallel kernels, such as is required by CUDA C, the programmer annotates the existing loops and data structures in the code so that an OpenACC compiler can target the code to different devices. For CUDA devices, the OpenACC compiler generates the kernels, creates the register and shared memory variables, and applies some of the performance optimizations that we have discussed in the previous chapters. The goal of OpenACC is to provide a programming model that is simple to use for domain scientists, maintains a single source code between different architectures, and is performance portable, meaning that code that performs well on one architecture will perform well on other architectures. In our experience, OpenACC also provides a convenient programming interface for highly skilled CUDA programmers to quickly parallelize large applications. The main communication channel between the user

and the compiler is the set of annotations on the source code. One can think of the user being a supervisor giving directions to the compiler as employees. Just like in any other managerial scenarios, having first-hand experience in the work that an employee does helps the manager to give better advice and directions. Now that we have learned and practiced the work that the OpenACC compiler does, we are ready to learn the effective ways to annotate the code for an OpenACC compiler.

> Read full chapter

From serial to parallel programming using OpenACC

Rob Farber, in Parallel Programming with OpenACC, 2017

The OpenACC Kernels Construct Compared to the Parallel Construct

OpenACC has two parallel computing constructs, the parallel construct used in the previous *accFilll_ex1.cpp* example and the kernels construct which will be demonstrated next.

Succinctly, a parallel construct in OpenACC tells the compiler that everything in the scope of the following region is a single parallel operation that will run in each thread. Adding a "loop" clause (per the preceding example) tells the compiler to try and parallelize everything inside the loop in each thread, which is the behavior OpenMP® programmers would expect. In CUDA terms, the parallel construct gets translated into a single CUDA kernel.

Safety tip: A common error is to forget to specify the loop directive (e.g., only specify #pragma acc parallel), which mistakenly tells the compiler that everything in the scope of the following code block will run in parallel, which means one **for** loop will run in each parallel thread! Also, it is easy to forget the "acc" in the pragma, in which case the loop will not get parallelized at all. These are but two of a myriad of reasons why it is important to check the compiler messages as well as verify that each parallel region actually runs in parallel on the device(s).

In contrast, a kernels construct gives the compiler much more flexibility to generate efficient parallel code for the targeted device(s) including combining loops into a single parallel kernel or creating multiple parallel kernels. It also places the responsibility on the compiler to ensure that it is safe to parallelize the loop, unlike parallel where it is told that it is safe to do so. This also means the compiler will tend

to be very cautious and sometimes require more information before it will parallelize certain loops. There are three steps to the kernels parallelization process:

1. Identify the loops that can be executed in parallel.
2. Map that abstract loop parallelism onto concrete hardware parallelism (e.g., threads that can run on a multicore processor or into to appropriate parallel configuration for a GPU).
3. Have the compiler generate and optimize the actual code to implement the selected parallelism mapping.

The following example, *accFill_ex2.cpp* utilizes the kernels construct to perform both the fill operation and calculation of the sum in parallel on the OpenACC device. Use of the kernels construct in this example eliminates the need to transfer the array as all computations will be performed on the device (Fig. 4).

Fig. 4. A kernels based data-parallel fill example accFill_ex2.cpp.

Note that the logic of the C++ code is identical between the two examples, but the OpenACC pragmas are different.

- accTask_ex1.cpp: #pragma acc parallel loop copyout(status[0:nCount])
- accTask_ex2.cpp: #pragma acc kernels create(status[0:nCount])

In addition, curly brackets, "{" and "}" were added to define the scope of the kernels construct. A create clause was also used to allocate space for the status array on the OpenACC device.

A vector reduction was specified via the second pragma "#pragma acc loop vector reduction(+:sum)."

Very simply, the reduction specified in the code sums all the values of in the **status** array using vector instructions when the target device supports them. More precisely, the reduction() clause takes an operator (in this case "+") and one or more scalar variables. Our example uses the variable **sum**. At the end of the OpenACC region the parallel result is combined with the value of the original variable. This is why **sum** must be initialized to some value (in this case zero) before the reduction else undefined behavior will result.

The vector clause tells the OpenACC that it can exploit vector parallelism.

Succinctly, vector instructions use hardware to effectively (from a software point of view) perform a number of operations at the same time, thus it is another form of parallelism. Each core in a modern multicore × 86 processor can issue vector instructions to a hardware vector units (or multiple per-core vector units) to perform multiple data-parallel operations at the same time. The AVX-512 vector instruction set, for example, is the current longest vector instruction set provided on ×86 processors that can perform up to 16 concurrent single-precision, 32-bit floating-point operations per instruction call. The end result is an up to 16× performance multiplier per vector unit. This can result in a large overall performance gain when all the vector units on all the cores in a high-end processor are fully utilized. GPUs utilize SIMD (Single Instruction Multiple Data) instructions to achieve a similar performance multiplier effect, except that vectorization naturally occurs in hardware across a group of threads referred to as a warp (CUDA terminology for a set of 32 threads) rather than through the explicit issuance of vector instructions by the programmer or compiler.

A schematic representation showing the performance benefits of vector and parallel programming on multicore processors is shown in Fig. 5.

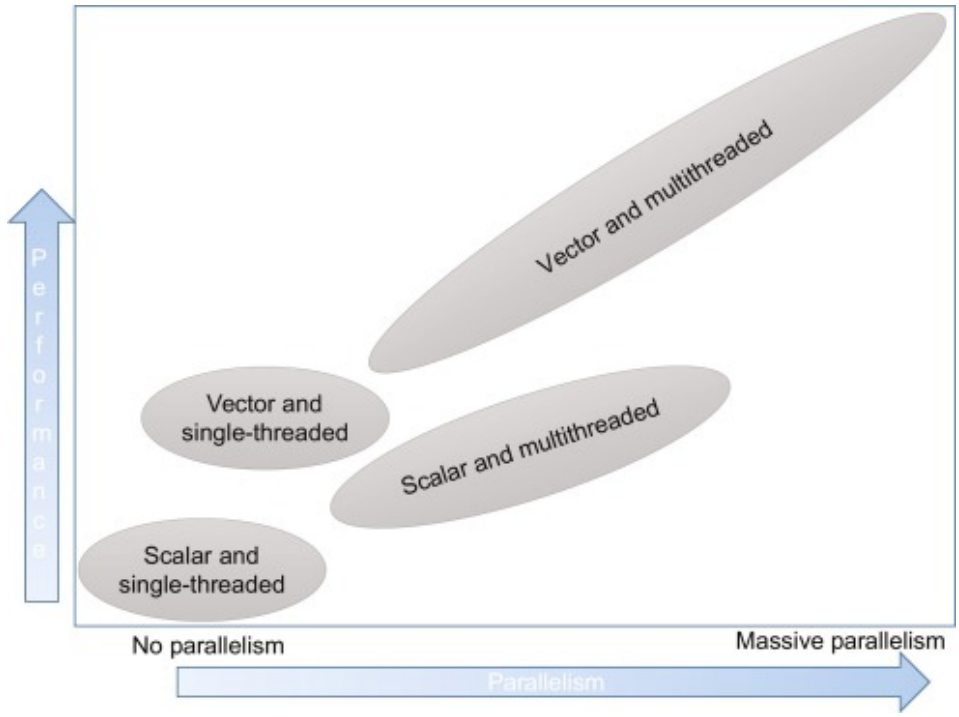

Fig. 5. Multicore parallel and vector performance.

> Read full chapter

Performance Evaluation

José Duato, ... Lionel Ni, in Interconnection Networks, 2003

9.12.2 Optimizations in the Software Messaging Layer

In this section we present some performance evaluation results showing the advantages of establishing virtual circuits between nodes, and caching and reusing those circuits. This technique is referred to as *virtual circuit caching* (VCC). VCC is implemented by allocating buffers for message transmission at source and destination nodes before data are available. Those buffers are reused for all the subsequent transmissions between the same nodes, therefore reducing software overhead considerably. A directory of cached virtual circuits is kept at each node. The evaluation results presented in this section are based on [81].

The simulation study has been performed using a time-step simulator written in C. The simulated topology is a k-ary n-cube. A simulator cycle is normalized to the time to transmit one flit across a physical link. In addition, the size of one flit is 2 bytes. Since the VCC mechanism exploits communication locality, performance evaluation was performed using traces of parallel programs. Traces were gathered for several parallel kernels executing on a 256-processor system.

The communication traces were collected using an execution-driven simulator from the SPASM toolset [323]. The parallel kernels are the (1) broadcast-and-roll parallel matrix multiply (MM), (2) a NAS kernel (EP), (3) a fast Fourier transform (FFT), (4) a Kalman filter (Kalman), and (5) a multigrid solver for computing a 3-D field (MG). The algorithm kernels display different communication traffic patterns.

Locality Metrics

The following metric attempts to capture communication locality by computing the average number of messages that are transmitted between a pair of processors:

This metric gives a measure of the number of messages that can use the same circuit, which as a result may be worth caching. However, this measure does not incorporate any sense of time; that is, there is no information about when each of the individual messages are actually sent relative to each other. In analyzing a trace for communication locality, we can compute the expression above over smaller intervals of time and produce a value that would be the average message density per unit of time. If parallel algorithms are structured to incorporate some temporal locality (i.e., messages from one processor to another tend to be clustered in time), then this measure will reflect communication locality. A trace with a high value of message density is expected to benefit from the use of the cacheable channels. Table 9.3 shows the computed values of the average message density for the traces under study. The traces with a value of 0.0 (EP, FFT, MG) possess no locality. They have no (source, destination) pairs that send more than one message to each other. The other traces have nonzero values, implying that reuse of circuits is beneficial.

Table 9.3. Message density for various traces.

Communication Trace	*Message Density*
EP	0.0
FFT	0.0
Kalman	0.848
MG	0.0
MM	0.466

The Effects of Locality

An uncached message transmission was modeled as the sum of the messaging layer overhead (including the path setup time) and the time to transmit a message. The overhead includes both the actual system software overhead and the network interface overhead. For example, this overhead was measured on an Intel Paragon, giving an approximate value of 2,600 simulator cycles. However, the selected value

for the overhead was 100 simulator cycles, corresponding to the approximate reported measured times for Active Message implementations [105]. A cached message transmission is modeled as the sum of the time to transmit the message (the actual data transmission time) and a small overhead to model the time spent in moving the message from user space to the network interface and from the network interface back into the user space. In addition, there is overhead associated with each call to set up a virtual circuit. This overhead is equivalent to that of an uncached transmission. There is also overhead associated with the execution of the directive to release a virtual circuit.

The effect of VCC is shown in Table 9.4. The differences are rather substantial, but there are several caveats with respect to these results. The VCC latencies do not include the amortized software overhead, but rather only network latencies. The reason is the following. The full software overhead (approximately 100 cycles) is only experienced by the first message to a processor. Subsequent messages experience the latencies shown in Table 9.4. Moreover, if virtual circuits are established before they are needed, it is possible to overlap path setup with computation. In this case, the VCC latencies shown in Table 9.4 are the latencies actually experienced by messages. Since these traces were simulated and optimized by manual insertion of directives in the trace, almost complete overlap was possible due to the predictable nature of many references. It is not clear that automated compiler techniques could do nearly as well, and the values should be viewed in that light. The VCC entries in the table are the average experienced latencies without including path setup time. The non-VCC latencies show the effect of experiencing overhead with every transmission. Although EP, FFT, and MG do not exhibit a great degree of locality, the results exhibit the benefit of overlapping path setup with computation. Finally, the overhead of 100 cycles is only representative of what is being reported these days with the Active Message [105] and the Fast Message [264] implementations. Anyway, the latency reduction achieved by using VCC in multicomputers is higher than the reduction achieved by other techniques.

Table 9.4. The effect of virtual circuit caching. Latencies are measured in cycles.

Program	*EP*	*FFT*	*Kalman*	*MG*	*MM*
VCC Latency	32.27	30.17	47.00	32.27	27.26
Non-VCC Latency	133.26	137.64	147.99	133.42	174.28

> Read full chapter

Addressing hardware reliability challenges in general-purpose GPUs

J. Tan, X. Fu, in Advances in GPU Research and Practice, 2017

2 GPGPUs Architecture

Fig. 1A shows an overview of the state-of-the-art GPUs [1]. They consist of a scalable number of in-order streaming multiprocessors (SM) that can access multiple memory controllers via an on-chip interconnection network. The GPU device has its own off-chip external memory (e.g., global memory) connected to the on-chip memory controllers. Fig. 1B illustrates a zoom-in view of the SM [22]. It contains the warp scheduler, register files (RF), streaming processors (SP), constant cache, texture cache, shared memory, and so on.

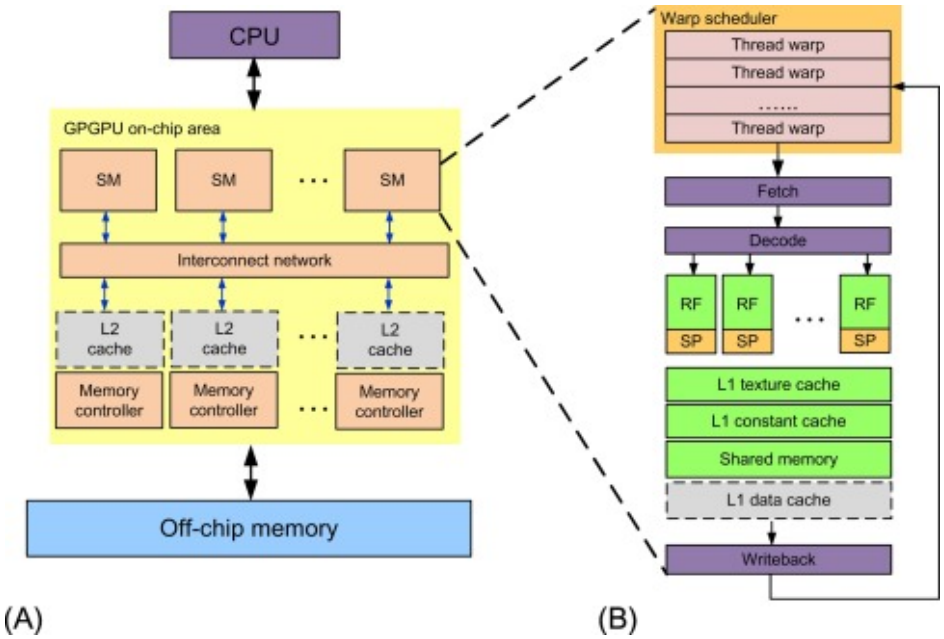

Fig. 1. General-purpose computing on graphic processing units (GPGPU) architecture. (A) An overview. (B) SM microarchitecture.

To facilitate GPGPUs' application development, several programming models have been developed (e.g., NVIDIA CUDA [1], AMD Stream [2], OpenCL [3]). In this chapter, we take the NVIDIA CUDA programming model as an example; the basic concepts will hold for most programming models. In CUDA, the GPU is treated as a coprocessor that executes highly parallel kernel functions launched by the CPU. The kernel is composed of a grid of lightweight threads; a grid is divided into a set of blocks (referred to as cooperative thread arrays [CTA] in CUDA); each block is composed of hundreds of threads. Threads are distributed to the SMs at the granularity of blocks, and threads within a single block communicate via shared memory and synchronize at a barrier if needed. Per-block resources, such as registers files, shared memory, and thread slots in an SM, are not released until

all the threads in the block finish execution. More than one block can be assigned to a single SM concurrently if there are sufficient per-block resources.

Threads in the SM execute in the SIMD fashion. A number of individual threads (e.g., 32 threads) from the same block are grouped together, which is called a warp. In the pipeline, threads within a warp execute the same instruction but with different data values. Fig. 1B also presents the details of SM microarchitecture. Each SM interleaves multiple warps (e.g., 32) on a cycle-by-cycle basis; the warp scheduler holds those warps, and at every cycle, it selects a warp with a ready instruction (i.e., the same instruction from all the threads within the warp are ready to execute) to feed the pipeline. The execution of a branch instruction in the warp may cause warp divergence when some threads jump while others fall through at the branch. Threads in a diverged warp have to execute in serial fashion, which greatly degrades the performance. Immediate postdominator reconvergence [23] has been widely used to handle the warp divergence. Recently, several mechanisms, such as dynamic warp formation (DWF) [24] and thread block compaction [22], have been applied to further improve the efficiency of branch handling. Because of the SIMD lock-step execution mechanism, a long-latency off-chip memory access from one thread can stall all the threads within a warp, and the warp cannot proceed until all the memory transactions complete. The load/store requests issued by different threads can get coalesced into fewer memory requests according to the access pattern. Memory coalescing improves performance by reducing the requests for memory access.

> Read full chapter

HSA Simulators

Y.-C. Chung, ... R. Ubal, in Heterogeneous System Architecture, 2016

9.1.3 HSAIL Host HSA

To remove the need for the host code, Multi2Sim allows users to launch HSA execution directly from an HSAIL program. This allows users to focus on learning HSAIL and enables them to debug an HSA program without the overhead of writing the host code, which can take longer than just writing the simple kernel itself. Multi2Sim implements a kernel launch and execution. First, it launches the kernel automatically as soon as the simulator starts. Second, the whole HSA runtime system [4] is supported by Multi2Sim in HSAIL via a set of runtime functions and a set of virtual device drivers. Although the HSA Foundation provides Okra [5], which is a lightweight interface to launch kernels on a simulator, Multi2Sim simulates the

system runtime, allowing the user to capture the interaction between devices and provide more information about the runtime execution.

9.1.3.1 Program entry

The main kernel is defined as:

kernel &main (kernarg_u32 %argc, kernarg_u64 %argv)

Execution starts with a kernel named &main. Main has two arguments: %argc and %argv. This standard C style program interface is utilized. In detail, the %argc is an unsigned 32-bit long integer and stores the number of arguments, while the %argv is the flat address pointing to an array of argument strings (starting at the first character). These kernel arguments can be omitted if a developer decides not to use command-line arguments. When the simulation starts, the simulator would first create an architected queuing language (AQL) queue for the hosting CPU device. This automatically injects an AQL packet into the queue to launch the &main kernel. This kernel launch forms a grid on the main CPU device with only a single work-item on the grid. The execution of the main grid is serialized. However, by using runtime functions, the user can easily discover other devices running in the simulated machine and launch parallel kernels.

9.1.3.2 HSA runtime interception

The runtime system is defined as a set of functions. Users may need to call those runtime functions to add queues, dispatch kernels, or do other runtime-related tasks. There is a one-to-one mapping between the official runtime functions and Multi2Sim HSAIL runtime functions. The mapping is simple. To utilize runtime functions in Multi2Sim directly, we need to know the associated official runtime function. For example, if the official runtime function in c is:

uint64_t hsa_queue_add_write_index_relaxed

(hsa_queue_t * queue, uint64_t value);

then the corresponding Multi2Sim HSAIL function would be:

function &hsa_queue_add_write_index_relaxed

(arg_u64 %ret) (arg_u64 %queue, arg_u64 %value) {};

The basic rules are as follows:

- The name of the function is identical, except for the & added in front of the function name, which is required by the HSAIL specification.
- Input and output arguments keep the same type. The name of the argument can be any valid HSAIL variable name.

- Pointers are all represented by 64-bit integer addresses, regardless of the machine type. Users have to cast the type if they are using a narrower (e.g., 32-bit) machine.
- Users do not need to implement the function. Even if they write in the function body, the instructions will be ignored by the simulator.

Internally, the simulator does not actually execute the runtime functions. Instead, it intercepts their invocations and converts them into application binary interface (ABI) calls that are passed to the HSA virtual device driver. The driver performs the designated action and returns the result via memory. A special case occurs when the runtime function uses a callback function. The driver builds a stack frame for the callback function and returns to the emulation environment. After the callback function returns, the driver intercepts execution again and returns to the place where the runtime function was called.

9.1.3.3 Basic I/O support

Because Multi2Sim-HSA supports running a standalone HSA program, we needed to provide the capability to effectively interact with the program. We have provided limited I/O support for HSAIL in Multi2Sim-HSA. In real hardware, if the kernel wants to perform an I/O operation, it has to build an AQL packet and send it to the CPU. Simulators commonly handle I/O commands using system calls, as has been implemented in the SPIM MIPS simulator. However, a system call interface is not presently available in the current HSAIL specification. Therefore, Multi2Sim supports a customized set of library-like functions for I/O. The general format is as follows:

function &m2s_action_TypeLength

(arg_TypeLength %input)

(arg_TypeLength %output)

where the action can be either print or read. The type and length can be an integer, unsigned integer, bit string, or float type, and is supported by HSAIL. For input functions, only the argument in the first set of parentheses would appear, while the argument is only used for output functions. The argument type and length must be identical to the type and length in the function name.

> Read full chapter

Parallel Computing

D.N.J. Clarke, ... S.A. Williams, in Advances in Parallel Computing, 1998

3.0 Templates applied to code

A variety of application codes have been used to test the RTIC methodology. One very useful set of examples are the kernels of the NAS Parallel Benchmarks[White94]. These relatively short abstracts from real application codes exhibit a variety of computational and communication patterns making a set of exemplars that are readily measured and understood [Hockney96], In addition they have already been implemented directly under PVM and thus allow comparisons to be made with the implementation using the RTIC methodology. The Embarrassingly Parallel Kernel (EP) of the NAS suite executes 2_{28} iterations of a loop in which random numbers are generated and tested. At the end of this computationally intensive process a small table of results is produced. There are no data or functional dependencies and thus little communication between processors. It thus acts to establish the peak performance of a particular platform. For PVM the parallelisation was straightforward as each slave process works independently on a subset of the random numbers supplied by the master to whom results would be returned. Initially the slaves are evaluated for their computational potential. In the RTIC version shown in Figure 3, the EP interface follows the message order used in the PVM implementation. The slaves' computational power is evaluated (Num_Sample_Iter), random number indices sent by master to slaves (Start_and_End), slave results returned to master (Results_Msg) and slave computation time returned to master (My_Time). The template is based on an One to Many framework. The USE construct is used to declare that the slave messages should be sent to the master through the MS_CH channel. The random number indices are also sent from master to slave on the MS_CH channel. Finally the EXECUTABLE constructs state which executable task type is to be used to generate the master and slave tasks.

The Conjugate Gradient method (CG) benchmark approximates the smallest eigenvalue of a symmetric positive definite sparse matrix of order 14000. The main design issue is how to store the various work vectors. The NAS choice is to divide the vector into as many pieces as there are rows of processors, with each row processor holding an identical copy of that piece of the vector. For dot products each row processor is assigned a portion of the vector to compute, with the summation done by a single processor. The matrix-vector multiplication is achieved by row computation and communication. The RTIC implementation of CG is based on a Toricgrid framework, which is used to produce the dot products, multiplication and summation. The associated communication is mapped through the COMBINE construct which incorporates a One to Many framework in which the selected process is the master task providing the summation for the calculation. Further One to Many frameworks provide communication for the matrix-vector multiplication along each column, with one task from each column acting as master. More One

to Many frameworks provide the communication for the row summations of the resultant vectors, one task from each row acting as master.

The Integer Sort benchmark (IS) performs 10 rankings of 2_{23} integer keys in the range 0 -2_{19}. Initially the root processor assigns the same number of keys to each processor. For each key the processors calculate the distribution and return their result array to the root processor. In turn the root distributes a balanced range of keys to each processor for ranking. Each processor then sends the appropriate assigned keys and indices to every other processor. The number of incoming keys each processor receives is broadcast by them to all processors. Finally each processor ranks the keys into a bucket array and returns rank and index. This distributed ranking scheme clearly produces a lot of interprocess communication; on a cluster system it may reduce essentially to a communications system benchmark. In the RTIC implementation this complex inter processor communication is built from two One to Many frameworks and a single All to All framework. The first One to Many framework is used to send the initial number of keys to the slave processes. The second One to Many framework handles the load balancing of the work between the slaves. The All to All framework is employed to achieve the final ranking of the keys. Once more the COMBINE construct can solve the mapping problem in a straightforward way.

Table 1 summarisesthe results of a number of runs of the NAS benchmarks either encoded directly in PVM or using the RTIC system. They have been run both on a cluster of Sun Sparc workstations and on a Silicon Graphics Onyx 2 using the number of processors shown in the table. The RTIC implementation follows the PVM version of the benchmarks. For all three benchmarks the total time taken is not significantly different between the PVM and RTIC systems. Thus it is possible to have the convenience of the RTIC formulation without incurring additional overhead. In more detail the EP figures show a massive computation load relative to the communications in keeping with the application. For IS this is reversed and most time is spent in communications. Finally with CG about one third of the time is devoted to communications using the Sparc workstations.

Table 1. NAS kernel benchmark results for the RTIC systems on Sun SparcStation and Silicon Graphics Onyx2.

NAS Benchmark (processors)	Message passing system	Total Time (secs)		Communication Time (secs)		Total library calls		Library send and recv calls	
		Sparc	Onyx2	Sparc		Onyx2			
EP(8)	PVM	6500	135	1	0.4	39	10		
EP(8)	RTIC	6500	135	1	0.4	15	8		
IS(8)	PVM	2013	60	1920	52	75	23		
IS(8)	RTIC	2016	60	1923	52	28	18		
CG(4)	PVM	1264	43	460	40	63	21		

CG(4)	RTIC	1266	43	462	40	27	18

The entries in the last two columns illustrate some aspects of the significant simplification that the RTIC methodology introduces. The total library calls column is a count of all the library calls included in all the application tasks. Typically this count is reduced by over a half when the RTIC system is used. The last column is a count of the send and receive operations employed by the task code. The RTIC implementation shows a small reduction in sends and receives but the percentage of such operations in the total number of library calls has increased. The overall reduction in library calls is explained by the fact that many of the explicit PVM process management calls and also the pack and unpack messages primitives, are not needed since they are included automatically in the applications library. Analysis of the PVM implementation of the CG and IS benchmarks shows that some of the most difficult parts of the code to understand are concerned with the building of lists of task identifiers for subsequent use by the message passing primitives. The introduction of the framework and COMBINE abstractions mean that this issue is handled by the application library. A significant source of programming error has been eliminated.

> Read full chapter

NWChem

Edoardo Aprà, ... Michael Klemm, in High Performance Parallelism Pearls, 2015

Engineering an offload solution

One key design decision when porting new software to the Intel Xeon Phi co-processor is what mode of operation and which programming model to use. With the native or symmetric mode, the coprocessors are autonomous compute nodes that participate in the computation by executing MPI or GA ranks with or without participation of their respective hosts. In the offload model, some computation remains on the host, while some kernels along with their input data are transferred to the coprocessor devices for execution.

From our domain knowledge, we understand that the structure of the TCE CCSD(T) algorithm exhibits excellent opportunities to use the offload model. It contains several floating-point intensive and highly parallel kernels. There are also opportunities for data reuse, so that communication between host and coprocessor can largely be avoided. In addition, some of the computation in preliminary steps of the algorithm (e.g., evaluation of two-electron repulsion integrals) require a substantial tuning

effort if a native approach is used, whereas in the offload model we can keep these computations on the host. Finally, we also expect that GA achieves a higher message rate and communication bandwidth on the host than it would on coprocessor.

Because we prefer the offload solution over native execution and symmetric MPI across host and coprocessors, potential offload candidates have to be identified. In complex applications such as NWChem it is a major effort, if not virtually impossible, without a proper methodology in place that guides developers through the process of evaluating the potential of an offload solution versus the native mode and of finding offload candidates. The methodology also helps implement the offload solution and apply optimizations to it. Our methodology is inspired by the iterative top-down methodology of software optimization (Yasin, 2014), in which a hotspot analysis is combined with optimizations of the most intriguing bottlenecks in the application code.

Figure 17.4 gives a graphical summary of the analysis methodology. All starts with the selection of the benchmark that mimics the application behavior as closely as possible, but on a smaller, more manageable scale. An ideal benchmark runs from a few up to 15 min and triggers the same code regions as the production data. With the benchmark, we execute the application and collect information to produce a hotspots profile. The hotspots in the profile are potential offload candidates that we then select for a call-tree analysis, which will give us information about (potentially) existing common functions that can be used as a common anchor points for offloading. As the next step, one needs to determine whether an offload candidate is suitable for execution on the coprocessor. The next data point (potentially augmented with additional performance data such as memory bandwidth, vectorization potential, etc.) comes from the loop analysis. Here we take a closer look at a hotspot and locate any loops, their minimum, maximum, and average trip counts. We also add information from the compiler's vectorization report for more complex loops to gain insight into a loop's suitability for both vectorization and parallelization.

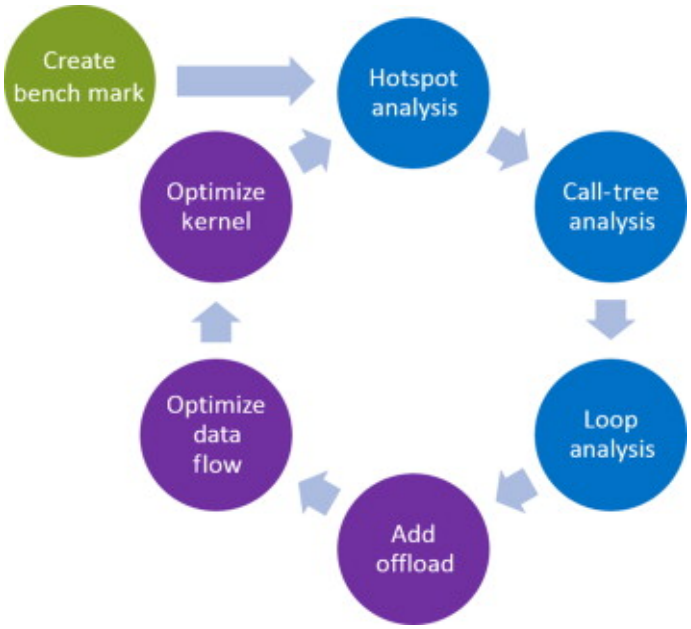

Figure 17.4. Iterative analysis methodology applied to NWChem.

The last three steps then deal with actual implementation work. After the analysis has been completed, we have all necessary data available to introduce appropriate offload pragmas (Intel Language Extensions for Offload), offload keywords (Intel Cilk Plus), or OpenMP target directives to the code. It is very likely that simply adding a offload for a single candidate leads to a suboptimal solution that will issue too many data transfers over the PCIe bus. Here, we take the call-tree analysis into account and use that information to hoist offloads and data transfers up in the call tree so that data transfers can be minimized. Finally, we work to optimize the individual offload regions to improve their performance on the target device.

We have applied the above methodology to the NWChem CCSD(T) method. As a benchmark, we use the uracil dimer benchmark as the input for the analysis. VTune Amplifier XE is our tool of choice to collect the hotpot profile for NWChem.

Figure 17.5 shows a screenshot of the GUI and various application hotspots. As can be seen, there is one big contribution (58%) to compute time by the comex_make_progress function that handles communication between NWChem processes and that consumes many cycles while waiting for incoming messages from other processes. In total, there are 38% of compute time that are attributed to a total 18 functions of the form sd_t_d*X*_*Y* (sd_t_d1_1 to sd_t_d1_9 and sd_t_d2_1 to sd_t_d2_9). All of them are called from the same anchor function (ccsd_d_doubles_l_2 in ccsd_t_doubles_l.F). Because of their exposure in the hotspot profile, these functions might be potential offload candidates.

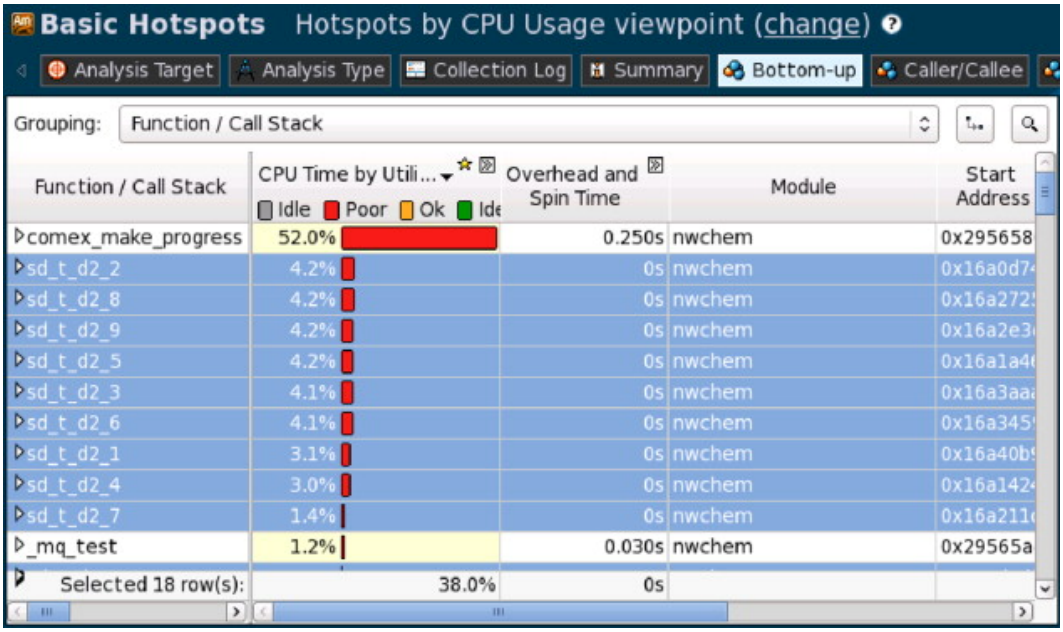

Figure 17.5. Result of the VTune Amplifier XE Hotspot analysis for NWChem's CCSD(T) method.

As our next step, we analyze the call tree that contains the sd_t_d*X*_*Y* functions. Figure 17.6 shows the corresponding screenshot of VTune Amplifier XE. The call-tree analysis reveals that all of the sd_t_d*X*_*Y* are called from a single function ccsd_d_doubles_l_2. As we will see in Section "Offload architecture," this function will be the main focus of the optimization work to avoid unnecessary data transfers to and from the coprocessor devices.

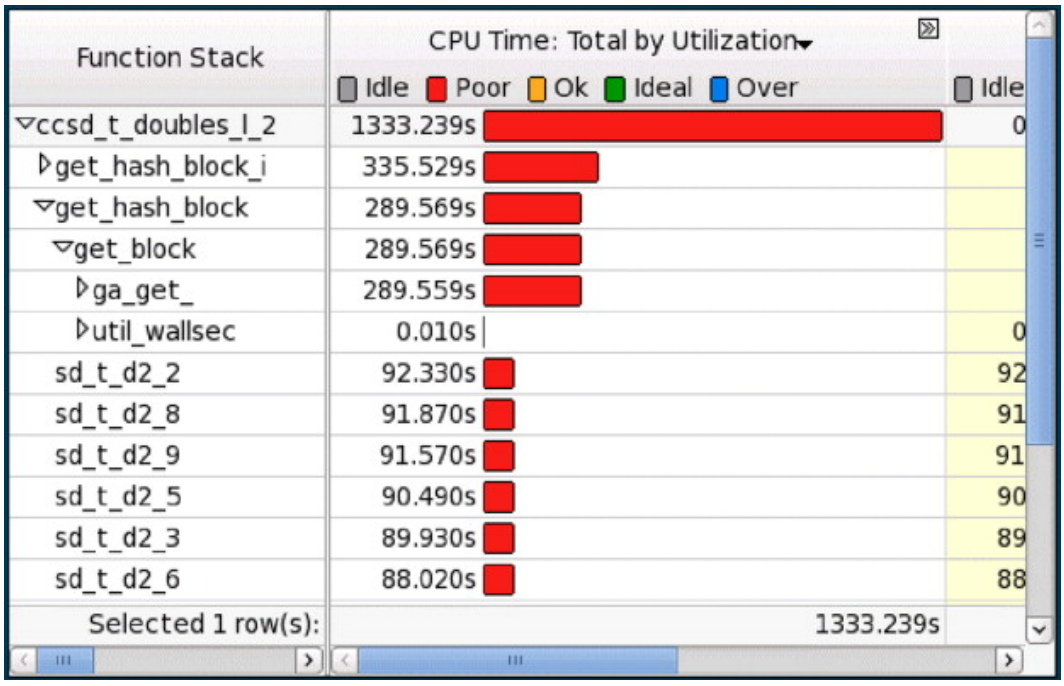

Figure 17.6. Call-tree analysis of the hotspots in Figure 17.5.

Taking the process further, we are now going to have a closer look at each individual sd_t_d*X*_*Y* kernel and make a final judgment about the suitability for offloading. As we know, the coprocessor requires both multithreading and SIMD vectorization to deliver optimal performance. Hence, we need to assess the kernels under these two metrics and determine which of the kernels will be a good offload candidate. Figure 17.7 shows the code of the sd_t_d1_1 kernel as a stand-in for all 18 kernels, which have the same basic code structure. Each kernel consists of seven perfectly nested loops that compute a very tight inner-most loop body. The trip counts of each of the loops are on the order of 20-30, which is not enough for trivial parallelization with OpenMP. Trivial (auto-)vectorization will also lead to a too low vectorization potential of about 80% only. However, the loops do not contain any loop-carried dependencies that would inhibit parallelization and vectorization, and are thus perfect candidates for offloading. Section "Kernel optimizations" will later present the optimizations applied to overcome the limited vectorization and parallelization potential.

Figure 17.7. Example kernel (sd_t_d1_1) of NWChem.

With the outcome of the analysis methodology we now have a plan on how to bring NWChem's CCSD(T) method to the Intel Xeon Phi coprocessor. The offload candidates identified are the kernels sd_t_d*X*_*Y* that are computationally very intensive fragments of code. The ccsd_d_doubles_l_2 function is the common anchor point of all the kernel invocations and we will introduce offload directives to the Fortran code to bring the kernels to execution and optimize the data transfers in the anchor point.

> Read full chapter

GPU-accelerated molecular dynamics clustering analysis with OpenACC

John E. Stone, ... Klaus Schulten, in Parallel Programming with OpenACC, 2017

Code Adaptation and Use of OpenACC Directives

At this stage, we already implemented efficient routines to copy selected atoms from VMD's internal AOS-layout atomic coordinate arrays from AOS and generate a new linearized and padded SOA-layout coordinate array for use in computing the dissimilarity matrix. We also have clean and performant reference implementations of the QCP inner product loop and the associated RMSD solver function. With these components in hand, we can proceed with final code transformations and addition of OpenACC compiler directives to achieve high performance on GPUs or other accelerators.

In the following, we show several versions of the dissimilarity matrix innermost loops adapted for acceleration with OpenACC. Since OpenACC directives apply to loops, it is important to show as much of the code as possible. To fit the code on a page it is necessary to leave out short code fragments in various places.

In general, the most important steps in adapting an algorithm for parallel execution on an accelerator are to minimize data transfers between the host Central Processing Units (CPUs) and accelerators, and to ensure that the organization of data in accelerator memory is conducive to high-bandwidth data-parallel access, both of which lead to good performance.

Fig. 5 combines the contents of the QCP inner product loop shown in Fig. 3, calls to the QCP Newton-Raphson solver, and the outermost loops over *i* and *j* that compute one triangle of the symmetric matrix of pairwise structure RMSDs—the dissimilarity matrix to be used in subsequent clustering analysis. The first OpenACC directives are added to the key loops as shown in Fig. 5. The most important OpenACC directives to add initially are kernels, copyin, and copy, which have been added to the outermost loops shown in Figs. 5 and 6.

```
  // abridged variable declarations for brevity ...
#pragma acc kernels copyin(crds[0:tsz]), copy(rmsdmat[0:msz])
  for (long j=0; j<framecount; j++) {
    float *crdx1 = crds + (j * 3L * framecrdsz);
    float *crdy1 = crdx1 + framecrdsz;
    float *crdz1 = crdx1 + framecrdsz*2;
#pragma acc loop
    for (long i=0; i<j; i++) {
      float *crdx2 = crds + (i * 3L * framecrdsz);
      float *crdy2 = crdx2 + framecrdsz;
      float *crdz2 = crdx2 + framecrdsz*2;
      // abridged zeroing of accumulators for brevity...
#pragma acc loop
      for (int l=0; l<cnt; l++) {
        x1 = crdx1[l];
        y1 = crdy1[l];
        z1 = crdz1[l];

        G1 += x1*x1 + y1*y1 + z1*z1;

        x2 = crdx2[l];
        y2 = crdy2[l];
        z2 = crdz2[l];

        G2 += x2*x2 + y2*y2 + z2*z2;

        a0 += x1 * x2;
        a1 += x1 * y2;
        a2 += x1 * z2;

        a3 += y1 * x2;
        a4 += y1 * y2;
        a5 += y1 * z2;

        a6 += z1 * x2;
        a7 += z1 * y2;
        a8 += z1 * z2;
      }

      double A[9];  A[0]=a0; /* abridged... */ A[8]=a8;
      double E0 = (G1 + G2) * 0.5;

      float rmsd;
      SolveRMSD(A, &rmsd, E0, cnt);
      rmsdmat[j*framecount + i]=rmsd;
    }
  }
```

Fig. 5. OpenACC kernel version 1 using multiple pointers for access to trajectory coordinates and 1D linearize storage of a 2D rectangular output matrix.

```
  // abridged variable declarations for brevity ...
#pragma acc kernels copyin(crds[0:tsz]), copy(rmsdmat[0:msz])
  for (long j=0; j<framecount; j++) {
    long x1addr = j * 3L * framecrdsz;
#pragma acc loop
    for (long i=0; i<j; i++) {
      long x2addr = i * 3L * framecrdsz;
      // abridged zeroing of accumulators for brevity...
#pragma acc loop
      for (int l=0; l<cnt; l++) {
        x1 = crds[l + x1addr];
        y1 = crds[l + x1addr + framecrdsz];
        z1 = crds[l + x1addr + framecrdsz*2];

        G1 += x1*x1 + y1*y1 + z1*z1;

        x2 = crds[l + x2addr];
        y2 = crds[l + x2addr + framecrdsz];
        z2 = crds[l + x2addr + framecrdsz*2];

        G2 += x2*x2 + y2*y2 + z2*z2;

        a0 += x1 * x2;
        a1 += x1 * y2;
        a2 += x1 * z2;

        a3 += y1 * x2;
        a4 += y1 * y2;
        a5 += y1 * z2;

        a6 += z1 * x2;
        a7 += z1 * y2;
        a8 += z1 * z2;
      }

      double A[9];  A[0]=a0; /* abridged... */ A[8]=a8;
      double E0 = (G1 + G2) * 0.5;

      float rmsd;
      SolveRMSD(A, &rmsd, E0, cnt);
      rmsdmat[j*framecount + i]=rmsd;
    }
  }
```

Fig. 6. OpenACC kernel version 2 using a single pointer but multiple indices for access to trajectory coordinates and 1D linearized storage of a 2D rectangular output matrix.

The copyin OpenACC directive informs the compiler that it should copy the named host-side array onto the GPU prior to launching the parallel kernel, and that it is not necessary for the data to be copied back after the kernel is complete, for example, the data is either read-only within the GPU kernel, or any modifications made to the data can be discarded and are not needed by the host. The copy directive is similar, except that it copies the named array from the host before the GPU kernel executes, and also copies it back to the host after completion. This is necessary for data that are written by the GPU and are ultimately needed on the host. In the case of the dissimilarity matrix calculation, the GPU kernels only write to a triangular region of

the array (in the case of a fully symmetric matrix storage scheme), so the matrix is zero-filled on the host and must be copied to and from the GPU so that any unwritten matrix elements are properly set to zero.

The kernels directive instructs the compiler that it should automatically analyze the loop nest contained by the *j* loop and look for opportunities to parallelize the loops contained in the loop nest. The kernels directive is an easy way to begin parallelizing a code initially, since it pushes the parallelization responsibility entirely onto the compiler, though in many cases the compiler needs to be explicitly informed which loops are independent and therefore safe to parallelize. Similarly, the loop directive guides the compiler to attempt to vectorize the loops that are so-labeled, using automatic heuristics to determine appropriate vector sizes, thread block sizes, and so on.

The function SolveRMSD called near the bottom of the listing in Fig. 5 refers to the QCP Newton-Raphson solver, which must be annotated with the OpenACC routine seq directives, to allow it to be run within a single accelerator thread. The use of the routine seq directives for SolveRMSD is shown in Fig. 7. The routine directive is also used to specify functions that should be parallelized, although that case does not apply to the algorithms involved in this chapter.

```
// QCP Newton–Raphson solver to compute the minimum RMSD
// given the ten inner product sums A[] required for
// construction of the 4x4 key matrix.
#pragma acc routine seq
int SolveRMSD(double *A, float *rmsd, double *E0, int cnt) {
  // body of the Newton–Raphson solver ommitted for brevity...
}
```

Fig. 7. OpenACC routine directive used to indicate that the QCP Newton-Raphson RMSD solver can be called within an individual thread on a GPU or another accelerator.

The code written in Fig. 5 uses several pointers for access to the coordinate arrays, which can present some difficulties for some OpenACC compilers. To make the code easier for the compiler to autovectorize, indexing arithmetic can be used on a single array pointer, as shown in Fig. 6. The use of indexing arithmetic in place of several pointers can, in some cases, provide a minor performance benefit since on many architectures, pointers are 64-bit quantities. The integer types used for index calculations can sometimes safely use smaller types that consume fewer registers, which can improve GPU thread occupancy (the number of threads that can be scheduled concurrently across the entire GPU). Although the use of a single array pointer simplifies the compiler's task of proving loop independence, further code changes are still needed.

The QCP inner product loop *l* ultimately performs data-parallel sum reductions. While the Portland Group (PGI) compilers automatically recognize the existence

of the sum reductions and handle them accordingly, some compilers need to be explicitly informed of the need to perform sum reductions on a0 through a8, and G1 and G2. For this purpose, OpenACC provides the reduction directive, which can be used to annotate sum reductions as: reduction(+:a0),reduction(+:a1), and so on. We have omitted the explicit annotation of sum reduction directives in the code listings since they are not needed by the PGI compilers. The reduction directives may be required for correct code generation by other compilers.

The output array rmsdmat is accessed by computing a linear index from the two *i* and *j* loop variables, but this causes a problem for many compilers that consider such an equation to pose a so-called "loop-carried dependency." If executed in parallel, loop-carried dependencies create a potential for output conflicts among loop iterations. To eliminate the loop-carried dependency we must either reformulate rmsdmat to use a two-dimensional array rather than a 1D linearized array, or we can change the loop structure to eliminate the *i* and *j* loops replacing them with a single loop *k* that operates in the linearized index space of rmsdmat. For our purposes it turns out to be much simpler to use the second approach, affording the opportunity to convert the code to use a more compact linearized representation of the upper- or lower-triangular portion of the dissimilarity matrix, thereby reducing the memory footprint. The cost of this approach is that *i* and *j* must be computed from the linearized index *k*, which requires a square root and other arithmetic to solve the quadratic equation that relates the indices *i* and *j* to *k* for the triangle of interest. The code shown in Fig. 8 implements this approach, which is the first version of the code that achieves effective parallelization on the GPU.

```
void rmsdmat_qcp_acc(int cnt, int padcnt, int framecrdsz,
                     int framecount, const float * restrict crds,
  // abridged function contents for brevity ...
  long i, j, k;
#pragma acc kernels copyin(crds[0:tsz]), copy(rmsdmat[0:msz])
  for (k=0; k<((framecount*(framecount-1))/2); k++) {
    acc_idx2sub_tril(long(framecount-1), k, &i, &j);
    long x1addr = j * 3L * framecrdsz;
    long x2addr = i * 3L * framecrdsz;

#pragma acc loop
    for (long l=0; l<cnt; l++) {
      x1 = crds[l + x1addr];
      y1 = crds[l + x1addr + framecrdsz];
      z1 = crds[l + x1addr + framecrdsz*2];

      G1 += x1*x1 + y1*y1 + z1*z1;

      x2 = crds[l + x2addr];
      y2 = crds[l + x2addr + framecrdsz];
      z2 = crds[l + x2addr + framecrdsz*2];

      G2 += x2*x2 + y2*y2 + z2*z2;

      a0 += x1 * x2;
      a1 += x1 * y2;
      a2 += x1 * z2;

      a3 += y1 * x2;
      a4 += y1 * y2;
      a5 += y1 * z2;

      a6 += z1 * x2;
      a7 += z1 * y2;
      a8 += z1 * z2;
    }

    double A[9];  A[0]=a0; /* abridged... */ A[8]=a8;
    double E0 = (G1 + G2) * 0.5;

    float rmsd;
    SolveRMSD(A, &rmsd, E0, cnt);
    rmsdmat[k]=rmsd; // store linearized triangular matrix
  }
```

Fig. 8. OpenACC kernel version 3 using a loop over the linearized index of the triangular matrix element, with coordinates accessed through a single pointer but multiple indices and 1D linearized storage of a 2D triangular output matrix.

Although the code modifications and OpenACC directives added in Fig. 8 achieves a high degree of parallelization, there is typically still room for improving performance further by providing the compiler with further guidance on the best vector size to use for a particular target accelerator architecture. In Fig. 9, we explicitly added the vector directive and provided guidance to the compiler to use a vector size of 256 by adding vector(256) to the existing loop directive. The larger vector size of 256 promotes better hardware utilization on contemporary GPUs, and has the effect of reducing the total number of thread blocks in-flight at once. The larger vector

size also reduces the number of memory accesses to disparate coordinate frames competing for very small on-chip caches.

```
void rmsdmat_qcp_acc(int cnt, int padcnt, int framecrdsz,
                     int framecount, const float * restrict crds,
  // abridged function contents for brevity ...
  long i, j, k;
#pragma acc kernels copyin(crds[0:tsz]), copy(rmsdmat[0:msz])
  for (k=0; k<((framecount*(framecount-1))/2); k++) {
    acc_idx2sub_tril(long(framecount-1), k, &i, &j);
    long x1addr = j * 3L * framecrdsz;
    long x2addr = i * 3L * framecrdsz;

#pragma acc loop vector(256)
    for (long l=0; l<cnt; l++) {
    // abridged for brevity ...

    rmsdmat[k]=rmsd; // store linearized triangular matrix
  }
}
```

Fig. 9. OpenACC kernel version 4, identical to version 3 except that the innermost loop has been annotated with the additional vector size directive which can be used to tune performance for a particular problem size or for characteristics of the target accelerator hardware.

> Read full chapter