Software Development Kit for Multicore Acceleration
Version 3.0

Accelerated Library Framework for Hybrid-x86 Programmer's Guide and API Reference Version 1.0 DRAFT

SC33-8406-00

Software Development Kit for Multicore Acceleration
Version 3.0

Accelerated Library Framework for Hybrid-x86 Programmer's Guide and API Reference Version 1.0 DRAFT

SC33-8406-00

Note
Before using this information and the product it supports, read the information in "Notices" on page 141.

Edition notice

This edition applies to Beta Version of the Software Development Kit for Multicore Acceleration Version 3.0 (program number 5724-S84) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this publication

This programmer's guide provides detailed information regarding the use of the Accelerated Library Framework APIs. It contains an overview of the Accelerated Library Framework, detailed reference information about the APIs, and usage information for programming with the APIs.

This book addresses the ALF implementation for Hybrid architecture.

For information about the accessibility features of this product, see Appendix E, "Accessibility features," on page 139.

Who should use this book

This book is intended for use by accelerated library developers and compute kernel developers.

Related information

See "Related documentation" on page 145.

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this publication, send your comments using Resource Link™ at http://www.ibm.com/servers/resourcelink. Click **Feedback** on the navigation pane. Be sure to include the name of the book, the form number of the book, and the specific location of the text you are commenting on (for example, a page number or table number).

Part 1. ALF overview

This section covers the following topics:

- A description of what is ALF is, see Chapter 1, “What is ALF?,” on page 3 and Chapter 2, “Overview of ALF external components,” on page 5
- What you can use ALF for, see Chapter 3, “When to use ALF,” on page 7
- ALF-specific concepts, see Chapter 4, “Concepts,” on page 9

Chapter 1. What is ALF?

The Accelerated Library Framework (ALF) provides a programming environment for data and task parallel applications and libraries. The ALF API provides you with a set of interfaces to simplify library development on heterogenous multi-core systems. You can use the provided framework to offload the computationally intensive work to the accelerators. More complex applications can be developed by combining the several function offload libraries. You can also choose to implement applications directly to the ALF interface.

ALF supports the multiple-program-multiple-data (MPMD) programming module where multiple programs can be scheduled to run on multiple accelerator elements at the same time.

The ALF functionality includes:

- Data transfer management
- Parallel task management
- Double buffering
- Dynamic load balancing for data parallel tasks

With the provided API, you can also create descriptions for multiple compute tasks and define their execution orders by defining task dependency. Task parallelism is accomplished by having tasks without direct or indirect dependencies between them. The ALF runtime provides an optimal parallel scheduling scheme for the tasks based on given dependencies.

ALF workload division

From the application or library programmer's point of view, ALF consists of the following two runtime components:

- A host runtime library
- An accelerator runtime library

The host runtime library provides the host APIs to the application. The accelerator runtime library provides the APIs to the application's accelerator code, usually the computational kernel and helper routines. This division of labor enables programmers to specialize in different parts of a given parallel workload.

ALF tasks

The ALF design enables a separation of work. There are three distinct types of task within a given application:

Application
: You develop programs only at the host level. You can use the provided accelerated libraries without direct knowledge of the inner workings of the underlying system.

Accelerated library
: You use the ALF APIs to provide the library interfaces to invoke the computational kernels on the accelerators. You divide the problem into the control process, which runs on the host, and the computational kernel,

which runs on the accelerators. You then partition the input and output into work blocks, which ALF can schedule to run on different accelerators.

Computational kernel
You write optimized accelerator code at the accelerator level. The ALF API provides a common interface for the compute task to be invoked automatically by the framework.

ALF runtime framework

The runtime framework handles the underlying task management, data movement, and error handling, which means that the focus is on the kernel and the data partitioning, and not on the direct memory access (DMA) list creation or management of the work queue.

The ALF APIs are platform-independent and their design is based on the fact that many applications targeted for multi-core computing follow the general usage pattern of dividing a set of data into self-contained blocks, creating a list of data blocks to be computed on the synergistic processing element (SPE), and then managing the distribution of that data to the various SPE processes. This type of control and compute process usage scenario, along with the corresponding work queue definition, are the fundamental abstractions in ALF.

Chapter 2. Overview of ALF external components

Within the ALF framework, a computational kernel is defined as an accelerator routine that takes a given set of input data and returns the output data based on the given input, see Figure 1. The input data and the corresponding output data are divided into separate portions, called work blocks. For a single task, ALF allows multiple computational kernels (of the same task) that run concurrently to process multiple work blocks in parallel, based on the available accelerator resources.

With the provided ALF API, you can also create descriptions for multiple compute tasks, and define their execution orders by defining their dependencies. Task parallelism is accomplished by having tasks without direct or indirect dependencies between them. The ALF runtime provides an optimal parallel scheduling scheme for the provided tasks based on the given dependencies.

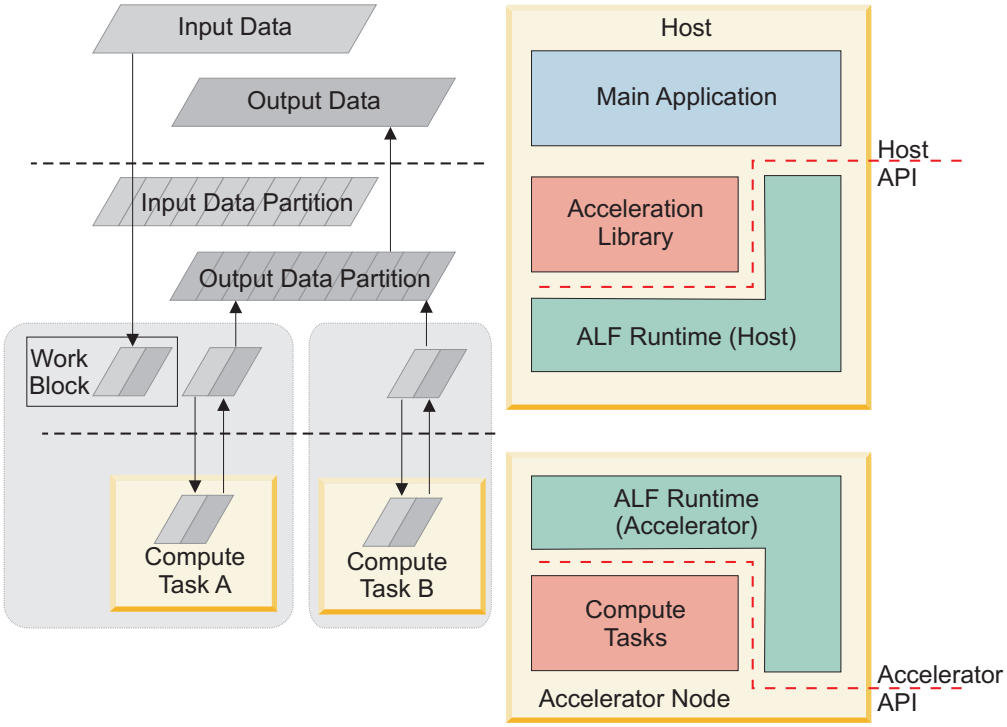

Figure 1. Overview of ALF

Chapter 3. When to use ALF

ALF is designed to help you to develop robust data parallel problems and task parallel problems. The following problem types are well suited to work on ALF:

- **Computationally intensive data-parallel problems**: The ALF API is designed to support data-parallel problems with the following characteristics:
  - Most of the parallel work focuses on performing operations on a large data set. The data set is typically organized into a common data structure, for example, an array of data elements.
  - A set of accelerators work collectively on the same data set, however, each accelerator works on a different partition of the data set. For ALF, the data set does not have to be regularly partitioned. Any accelerator can be set to work on any part of the data set.
  - The programs on the accelerators usually perform the same task on the data set.
- **Task-parallel problems**: The ALF API supports multiple tasks running on multiple accelerators at the same time. You can divide your application into subproblems and create one task for each subproblem. The ALF runtime can then determine the best way to schedule the multiple tasks on the available accelerators to get the most parallelism.

Certain problems can seem to be inherently serial at first; however, there might be alternative approaches to divide the problem into subproblems, and one or all of the subproblems can often be parallelized.

You need to be aware of the physical limitations on the supported platforms. If the data set of the problem cannot be divided into work blocks that fit into local storage, then ALF cannot be used to solve that problem.

Chapter 4. Concepts

The following sections explain the main concepts and terms used in ALF. It covers the following topics:

Computational kernel

A computational kernel is a user-defined accelerator routine that takes a given set of input data and returns the output data based on the given input. You should implement the computational kernel according to the function prototype definitions with the data in the provided buffers (see "Accelerator buffer management" on page 18). Then the computational kernel must be registered to the ALF runtime when the corresponding task descriptor is created.

The computational kernel is usually accompanied by four other auxiliary functions. The five of them forms a 5-tuple for a task as:

```
{
alf_accel_comp_kernel,
alf_accel_input_dtl_prepare,
alf_accel_output_dtl_prepare,
alf_accel_task_context_setup,
alf_accel_task_context_merge
}
```

Note: The above accelerator function names are used as conventions for this document only. You can provide your own function name for each of these functions and register the function name through the task descriptor service.

Based on the different application requirements, some of the elements in this 5-tuple can be NULL.

For more information about the APIs that define computational kernels, see "User-provided computational kernel APIs" on page 94.

Task descriptor

A task descriptor contains all the relevant task descriptions. To maximize accelerator performance, ALF employs a static memory allocation model per task execution on the accelerator. This means that ALF requires you to provide information about buffers, stack usage, and the number of data transfer list entries ahead of time. As well as accelerator memory usage information, the task descriptor also contains information about the names of the different user-defined accelerator functions and the data partition attribute.

The following information is used to define a task descriptor:

- Task context description
  - Task context buffer size
  - Task context entries: entry size, entry type
- Accelerator executable image that contains the computational kernel:
  - The name of the accelerator computational kernel function
  - Optionally, the name of the accelerator input data transfer list prepare function
  - Optionally, the name of the accelerator output data transfer list prepare function
  - Optionally, the name of the accelerator task context setup function
  - Optionally, the name of the accelerator task context merge function
- Work block parameter context buffer size
- Work block input buffer size
- Work block output buffer size
- Work block overlapped buffer size
- Work block number of data transfer list entries
- Task data partition attribute:
  - Partition on accelerator
  - Partition on host
- Accelerator stack size

For more information about the compute task APIs, see “Compute task API” on page 63.

Task

A task is defined as a ready-to-be-scheduled instantiation of a task description or you use the `num_instances` parameter in the task creation function (`alf_task_create`), to explicitly request a number of accelerators or let the ALF runtime decide the necessary number of accelerators to run the compute task. You can also provide the data for the context buffer associated with this particular task.

You can also register an event handler to monitor different task events, see “Task events” on page 12.

After you have created a task, you can create work blocks and enqueue the work blocks on to the working queue associated with the task. The ALF framework employs an immediate runtime mode. After a work block has been enqueued, if the task has no unresolved dependency on other tasks, the task is scheduled to process the work blocks.

For information about work blocks, see “Work blocks” on page 12.

Task finalize

After you have finished adding work blocks to the work queue, you must call `alf_task_finalize` function to notify ALF that there are no more work blocks for this particular task. A task that is not ″finalized″ cannot be run to the completion state.

Task dependency and task scheduling

In ALF programming model, task dependency is used to make sure multiple tasks can be run in a specific order when the order is critical. Some common dependency scenarios are listed here:

- Data dependency: where the output of one task is the input of another task
- Resource conflict: where the tasks share some common resources such as temporary buffers
- Timing: where the tasks have to follow a predetermined order

After you have created a task, you can use the function `alf_task_depends_on` to specify the task's dependency with an existing task. The ALF runtime considers a task's dependency and the number of requested accelerators for scheduling.

The ALF framework does not detect circular dependency. For a task that depends on other tasks, you must define the dependencies before any work block is added to the task. If a circular dependency is present in the task definitions, the application hangs. The ALF framework provides a debug option (at compile time) to check for circular dependencies, however, the ALF framework does not check for circular dependency during normal runtime.

A task that depends on other tasks cannot be processed until all the dependent tasks finish. Tasks are created in immediate mode. After a task has been created and its dependencies are satisfied, the task can be scheduled to run.

For an example of how to use task dependency, see “Task dependency example” on page 121.

Task instance

A task can be scheduled to run on multiple accelerators. Each task running on an accelerator is a task instance. If a task is created without the `ALF_TASK_ATTR_SCHED_FIXED` attribute, the ALF runtime can load and unload an instance of a task to and from an accelerator anytime.

The ALF runtime posts an event after a task instance is started on an accelerator or unloaded from an accelerator. You can choose to register an event handler for this event, see “Task events” on page 12.

Fixed task mapping

For task scheduling, you can explicitly require the runtime to start a fixed number of task instances for a specific task. This is known as fixed task mapping. To do this, you need to :

1. Provide the number of task instances at task creation time through the `alf_task_create` interface

2. Set the ALF_TASK_ATTR_SCHED_FIXED task attribute

In this case, the runtime makes sure all the task instances are started before work blocks are assigned to them.

Task context

Note: For more information, refer to "Task context buffer" on page 18.

A task context is used to address the following usage scenarios:

Common persistent data across work blocks

A task context can be used as common persistent referenced data for all work blocks in a task. This is especially useful for static input data, lookup tables, or any other input data that is common to all work blocks. Because the ALF runtime loads the task context to accelerator memory before any work block is processed, you can be assured that the common data is always there for the work blocks to use.

Reducing partial results across work blocks

A task context can be used to incrementally update the final result of a task based on each work block's computation. For these applications, the computational results of separate work blocks are the intermediate results. These intermediate results are stored in the task context. You can update the task context in the computational kernel as part of the work block computation. After all the work blocks have been processed, the ALF runtime applies a reduction step to merge the intermediate results of the task instances into a single final result using the provided alf_accel_task_context_merge function.

For an example about how to apply the concept of task context to find the maximum value or the minimum value of a large data set, see "Min-max finder example" on page 115.

Task events

The ALF framework provides notifications for the following task events:

- ALF_TASK_EVENT_READY - the task is ready to be scheduled
- ALF_TASK_EVENT_FINISHED - the task has finished running
- ALF_TASK_EVENT_FINALIZED - all the work blocks for the task have been enqueued alf_task_finalized has been called
- ALF_TASK_EVENT_INSTANCE_START - one new instance of the task starts
- ALF_TASK_EVENT_INSTANCE_END - one instance of the task ends
- ALF_TASK_EVENT_DESTROY - The task is destroyed explicitly

For information about how to set event handling, see alf_task_event_handler_register.

Work blocks

A work block represents an invocation of a task with a specific set of related input data, output data, and parameters. The input and output data are described by corresponding data transfer lists. The parameters are provided through the ALF APIs. Depending on the application, the data transfer list can either be generated on the host (host data partition) or by the accelerators (accelerator data partition).

Before it calls the compute task, and as the ALF accelerator runtime processes a work block it retrieves the parameters and the input data based on the input data transfer list to the input buffer in host memory. After it has invoked the computational kernel, the ALF accelerator runtime puts the output result back into the host memory. The ALF accelerator runtime manages the memory of the accelerator to accommodate the work block's input and output data. The ALF accelerator runtime also supports overlapping data transfers and computations transparently through double buffering techniques if there is enough free memory.

Single-use work block

A single-use work block is processed only once. A single-use work block enables you to generate input and output data transfer lists on either the host or the accelerator.

Multi-use work block

A multi-use work block is repeatedly processed up to the specified iteration count. Unlike the single-use work block, the multi-use work block does not enable you to generate input and output data transfer lists from the host process. For multi-use work blocks, all input and output data transfer lists must be generated on the accelerators each time a work block is processed by the ALF runtime. For each iteration of the multi-use work block, the ALF runtime passes the parameters, total number of iterations, and current iteration count to the accelerator data partition subroutines, and you can generate the corresponding data transfer lists for each iteration based on this information. See "Accelerator data partitioning" on page 17 for more information about single-use work blocks and multi-use work blocks.

Work block scheduling

This section describe work block scheduling. It covers the following:

Default work block scheduling policy

The ALF API supports multiple ways of assigning work blocks to task instances. By default, enqueued work blocks can be assigned to any of the task instances in any order. The ALF runtime tries to balance the load of the task instances to ensure that the task can complete in the shortest time. This means that task instances that start early or run faster may process more work blocks than those that start later or run slower. Figure 2 on page 14 shows an example of the default work block scheduling policy where task instances process work blocks at different rates.

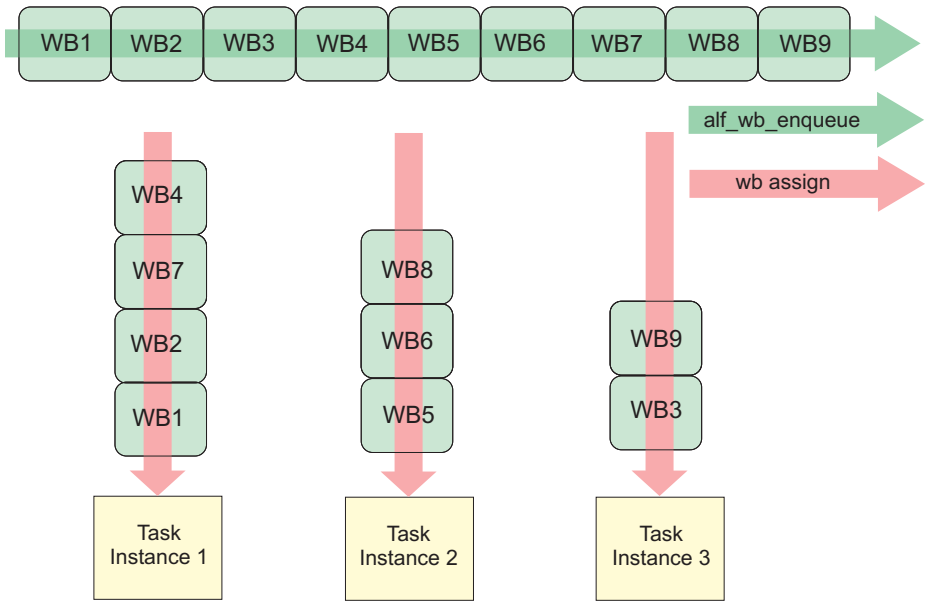

Figure 2. Default work block scheduling behavior

Cyclic distribution policy

You can enable cyclic work block distribution by setting the attributes `ALF_TASK_ATTR_WB_CYCLIC` and `ALF_TASK_ATTR_SCHED_FIXED` when you create the task. These attributes enable the work blocks to be assigned in a round robin order to a fixed number of task instances. You must provide the number of task instances in the `alf_task_create` function. The work blocks are assigned to the task instances in a cyclical manner in the order of the work blocks being enqueued through calling the function `alf_wb_enqueue`. Figure 3 on page 15 shows an example of cyclic work block distribution.

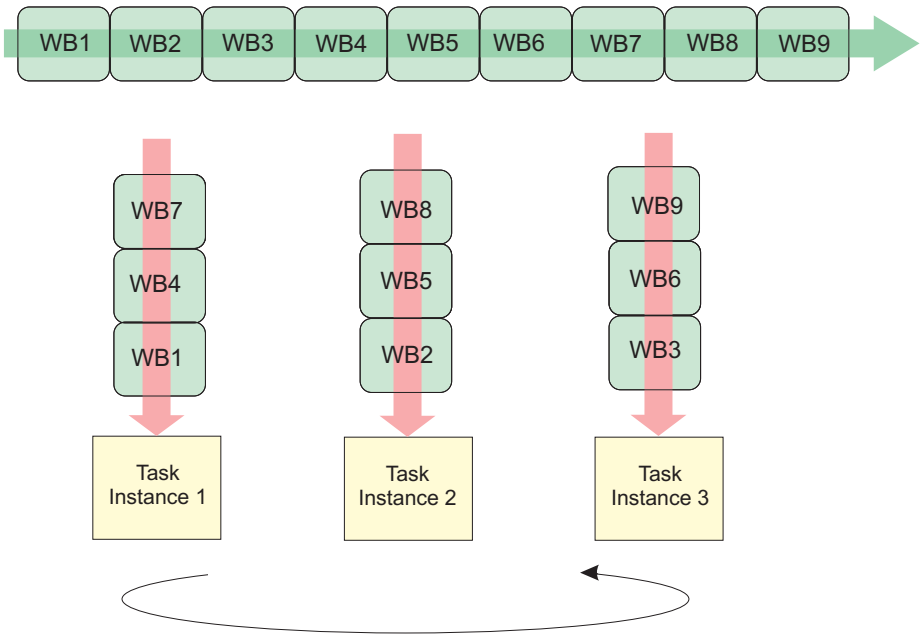

Figure 3. Cyclic work block distribution

Bundled distribution policy

The work blocks are assigned to the task instances in a group of `bundle_size` at a time in the order of the work blocks being enqueued through calling the function `alf_wb_enqueue`. All work blocks in a bundle are assigned to one task instance, and the order defined in `alf_wb_enqueue` is also preserved. You use the parameter `wb_dist_size` to specify the bundle size when you create the task. Bundled distribution can also be used together with the cyclic distribution to further control the work block scheduling. Figure 4 shows an example of the bundled distribution policy where task instances process work blocks at different rates.

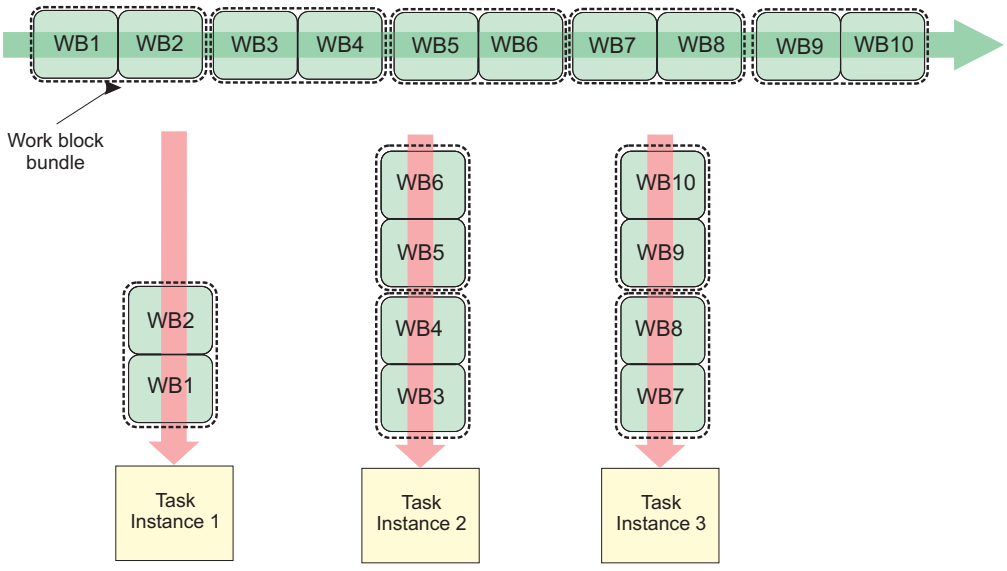

Figure 4. Bundled work block distribution

Data partitioning

An important part to solving data parallel problems using multiple accelerators is to figure out how to partition data across the accelerators. The ALF API does not automatically partition data, however, it does provide a framework so that you can systematically partition the data.

The ALF API provides the following different data partition methods:

- "Host data partitioning" on page 17
- "Accelerator data partitioning" on page 17

These methods are described in the following sections together with the Figure 5 on page 17.

Data transfer list

For many applications, the input data for a single compute kernel cannot be stored contiguously in the host memory. For example, in the case of a multi-dimensional matrix, the matrix is usually partitioned into smaller sub-matrices for the accelerators to process. For many data partitioning schemes, the data of the sub-matrices is scattered to different host memory locations. Accelerator memory is usually limited, and the most efficient way to store the submatrix is contiguously. Data for each row or column of the submatrix is put together in a contiguous buffer. For input data, they are gathered to the local memory of the accelerator from scattered host memory locations. With output data, the above situation is reversed, and the data in the local memory of the accelerator is scattered to different locations in host memory.

The ALF API uses data transfer list to represent the scattered data in the host memory. A data transfer list contains entries that consist of the data size and a pointer to the host memory location of the data. The data in the local memory of the accelerator is always packed and is organized in the order of the entries in the list. For input data, the data transfer list describes a data gathering operation. For output data, the data transfer list describes a scattering operation. See Figure 5 on page 17 for a diagram of a data transfer list.

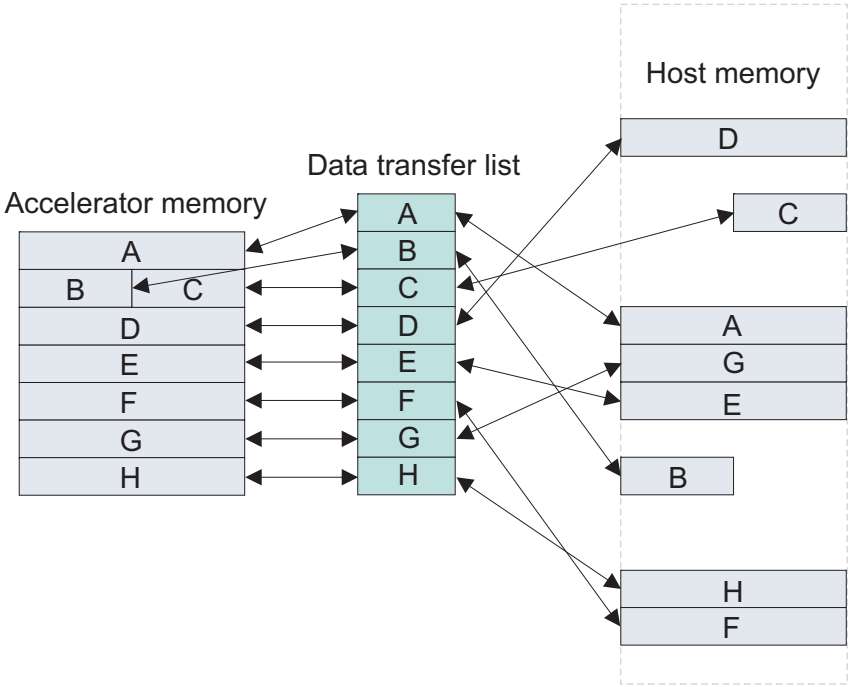

Figure 5. Data transfer list

To maximize accelerator performance, ALF employs a static memory allocation model per task execution on the accelerator. This means programmers need to explicitly specify the maximum number of entries a data transfer list in a task can have. This can be set through the `alf_task_desc_set_int32` function with the `ALF_TASK_DESC_NUM_DTL_ENTRIES` function.

Host data partitioning

You can use the provided APIs on the host to partition your applications' data . To do this, you build a data transfer list for the work blocks thru the `alf_wb_dtl_begin`, `alf_wb_dtl_entry_add`, and `alf_wb_dtl_end` APIs.

Partitioning data using the host APIs is simpler than partitioning data on the accelerator, (for more information, see "Accelerator data partitioning"). After you have identified the appropriate input and output data associated with each work block, the ALF runtime automatically generates the necessary data transfer lists and pulls the necessary data into the accelerator's memory for processing.

This method is particularly useful when the data associated with the work blocks is simple, and the host can keep up with generating the data partitioning information for all the accelerators.

Accelerator data partitioning

When the data partition schemes are complex and require a lot of computing resources, it can be more efficient to generate the data transfer lists on the accelerators. This is especially useful if the host computing resources can be used for other work or if the host does not have enough computing resources to compute data transfer lists for all of its work blocks.

Accelerator data partition APIs

Accelerated library developers must provide the `alf_accel_input_dtl_prepare` subroutine and the `af_accel_output_dtl_prepare` subroutine to do the data partition for input and output and generate the corresponding data transfer list. The `alf_accel_input_dtl_prepare` is the input data partitioning subroutine and the `alf_accel_output_dtl_prepare` is the output data subroutine.

Host memory addresses

The host does not generate the data transfer lists when using accelerator data partitioning, so the host addresses of input and output data buffers can be explicitly passed to the accelerator through the work block parameter and context buffer.

For an example, see "Matrix add - accelerator data partitioning example" on page 112

Accelerator buffer management

On the accelerator, the ALF accelerator runtime manages the data of the work blocks and the task for the compute kernel. You only need to focus on the organization of data and the actual computational kernel. The ALF accelerator runtime handles buffer management and data movement. However, it is still important that you have a good understanding of how each buffer is used and its relationship with the computational kernel.

To make the most efficient use of accelerator memory, the ALF runtime needs to know the memory usage requirements of the task. The ALF runtime requires that you specify the memory resources each task uses. The runtime can then allocate the requested memory for the task.

Buffer types

The ALF accelerator runtime code provides handles to the following different buffers for each instance of a task:

- "Task context buffer"
- "Work block parameter and context buffer" on page 19
- "Work block input data buffer" on page 20
- "Work block output data buffer" on page 20
- "Work block overlapped input and output data buffer" on page 20

Task context buffer

A task context buffer is used by applications that require common persistent data that can be referenced and updated by all work blocks. It is also useful for merging operations or all-reduce operations. A task is optionally associated with one task context buffer. You can specify the size of the task context buffer through the task descriptor creation process. If the size of the task context buffer is specified as zero (0) in the task descriptor, there is no task context associated with the any of the tasks created with that task descriptor.

The lifecycle of the task context is shown in Figure 6 on page 19. To create the task, you call the task creation function `alf_task_create`. You provide the data for the initial task context by passing a data buffer with the initial values. After the

compute task has been scheduled to be run on the accelerators, the ALF framework creates private copies of the task context for the task instance that is running.

You can provide a function to initialize the task context (`alf_accel_task_context_setup`) on the accelerator. The ALF runtime invokes this function when the running task instance is first loaded on an accelerator as shown in Figure 6 (a).

All work blocks that are processed by one task instance share the same private copy of task context on that accelerator as shown in Figure 6 (b).

When the ALF scheduler requests an accelerator to unload a task instance, you can provide a merge function (`alf_accel_task_context_merge`), which is called by the runtime, to merge that accelerator's task context with an active task context on another accelerator as shown in Figure 6 (c).

When a task is shut down and all instances of the task are destroyed, the runtime automatically calls the merge function on the task instances to merge all of the private copies of task context into a single task context and write the final result to the task context on host memory provided when the task is created, as shown in Figure 6 (d).

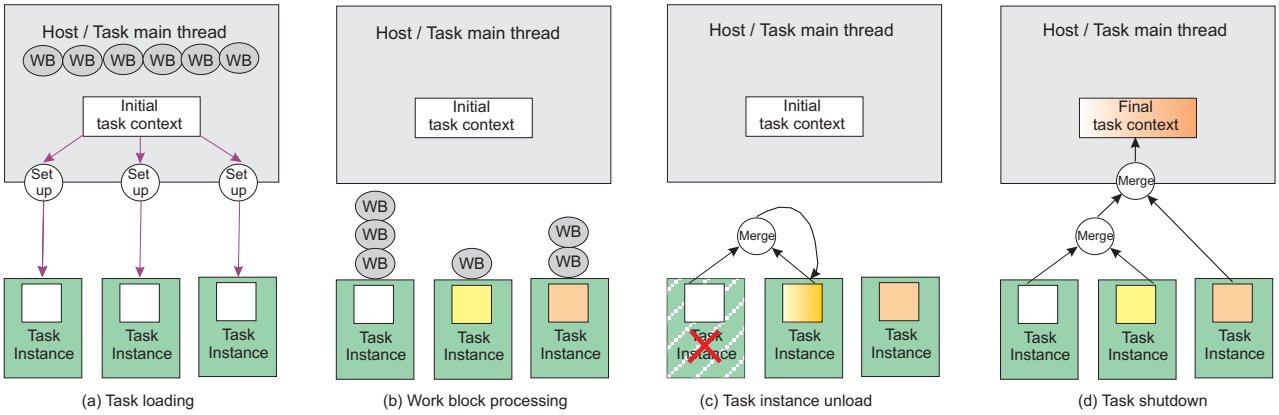

Figure 6. Task context buffer lifecycle

Work block parameter and context buffer

The work block parameter and context buffer serves two purposes:

- It passes work block-specific constants or reference-by-value parameters
- It reserves storage space for the computational kernel to save the data specific to one work block, which can be either a single-use work block or a multi-use work block

This buffer can be used by the following APIs:

- `alf_accel_comp_kernel`
- `alf_accel_input_dtl_prepare`
- `alf_accel_output_dtl_prepare`

The parameters are copied to an internal buffer associated with the work block data structure in host memory when the `alf_wb_add_parm` accelerator routine is invoked.

For more information, see “Modifying the work block parameter and context buffer when using multi-use work blocks” on page 22.

Work block input data buffer

The work block input data buffer contains the input data for each work block (or each iteration of a multi-use work block) for the compute kernel. For each iteration of the ALF computational kernel, there is a single contiguous input data buffer. However, the data for the input buffer can come from distinct sections of a large data set in host memory. These separate data segments are gathered into the input data buffer on the accelerators. The ALF framework minimizes performance overhead by not duplicating input data unnecessarily. When the content of the work block is constructed by `alf_wb_dtl_entry_add`, only the pointers to the input data chunks are saved to the internal data structure of the work block. This data is transferred to the memory of the accelerator when the work block is processed. A pointer to the contiguous input buffer in the memory of the accelerator is passed to the computational kernel.

For more information about data scattering and gathering, see “Data transfer list” on page 16.

Work block output data buffer

This buffer is used to save the output of the compute kernel. It is a single contiguous buffer in the memory of the accelerator. Output data can be transferred to distinct memory segments within a large output buffer in host memory. After the compute kernel returns from processing one work block, the data in this buffer is moved to the host memory locations specified by the `alf_wb_dtl_entry_add` routine when the work block is constructed.

Work block overlapped input and output data buffer

The overlapped input and output buffer (overlapped I/O buffer) contains both input and output data. The input and output sections are dynamically designated for each work block.

This buffer is especially useful when you want to maximize the use of accelerator memory and the input buffer can be overwritten by the output data.

For more information about when to use this buffer, refer to Chapter 7, “When to use the overlapped I/O buffer,” on page 31.

For an example of how to use the buffer, see “Overlapped I/O buffer example” on page 119.

Using work blocks and order of function calls per task instance on the accelerator

Based on the characteristics of an application, you can use single-use work blocks or multi-use work blocks to efficiently implement data partitioning on the accelerators. For a given task that can be partitioned into *N* work blocks, the following describes how the different types of work blocks can be used, and also the order of function calls per task instance based on a single instance of a the task on a single accelerator:

1. Task instance initialization (this is done by the ALF runtime)

2. Conditional execute: `alf_accel_task_context_setup` is only called if the task has context. The runtime calls it when the initial task context data has been loaded to the accelerator and before any work blocks are processed.
3. For each work block WB(k):
   a. If there are pending context merges, go to Step 4.
   b. For each iteration of a multi-use work block `i < N` (total number of iteration)
      1) `alf_accel_input_list_prepare(WB(k), i, N)`: It is only called when the task requires accelerator data partition.
      2) `alf_accel_comp_kernel(WB(k), i, N)`: The computational kernel is always called.
      3) `alf_accel_output_list_prepare(WB(k), i, N)`: It is only called when the task requires accelerator data partition.
4. Conditional execute: `alf_accel_task_context_merge` This API is only called when the context of another unloaded task instance is to be merged to current instance.
   a. If there are pending work blocks, go to Step 3.
5. Write out task context.
6. Unload image or pending for next scheduling.
   a. If a new task instance is created, go to Step 2.

For step 3, the calling order of the three function calls is defined by the following rules:

- For a specific single-use work block `WB(k)`, the following calling order is guaranteed:
  1. `alf_accel_input_list_prepare(WB(k))`
  2. `alf_accel_comp_kernel(WB(k))`
  3. `alf_accel_output_list_prepare(WB(k))`
- For two single-use work blocks that are assigned to the same task instance in the order of `WB(k)` and `WB(k+1)`, ALF only guarantees the following calling orders:
  – `alf_accel_input_list_prepare(WB(k))` is called before `alf_accel_input_list_prepare(WB(k+1))`
  – `alf_accel_comp_kernel(WB(k))` is called before `alf_accel_comp_kernel(WB(k+1))`
  – `alf_accel_output_list_prepare(WB(k))` is called before `alf_accel_output_list_prepare(WB(k+1))`
- For a multi-use work block `WB(k,N)`, it is considered as *N* single use work blocks assigned to the same task instance in the order of incremental iteration index `WB(k,0), WB(k, 1), ..., WB(k, N-1)`. The only difference is that all these work blocks share the same work block parameter and context buffer. Other than that, the API calling order is still decided by the previous two rules. See "Modifying the work block parameter and context buffer when using multi-use work blocks" on page 22.

Modifying the work block parameter and context buffer when using multi-use work blocks

The work block parameter and context buffer of a multi-use work block is shared by multiple invocations of the `alf_accel_input_dtl_prepare` accelerator function and the `alf_accel_output_dtl_prepare` accelerator function. Take care when you change the contents of this buffer. Because the ALF runtime does double buffering transparently, it is possible that the `current_count` arguments for succeeding calls to the `alf_accel_input_dtl_prepare` function, the `alf_accel_comp_kernel` function, and the `alf_accel_output_dtl_prepare` function are not strictly incremented when a multi-use work block is processed. Because of this, modifying the parameter and context buffer according to the `current_count` in one of the subroutines can cause unexpected effects to other subroutines when they are called with different `current_count` values at a later time.

Data set

An ALF data set is a logical set of data buffers. A data set informs the ALF runtime about the set of all data to which the task's work blocks refer. The ALF runtime uses this information to optimize how data is moved from the host's memory to the accelerator's memory and back.

Figure 7 shows how data sets are used.

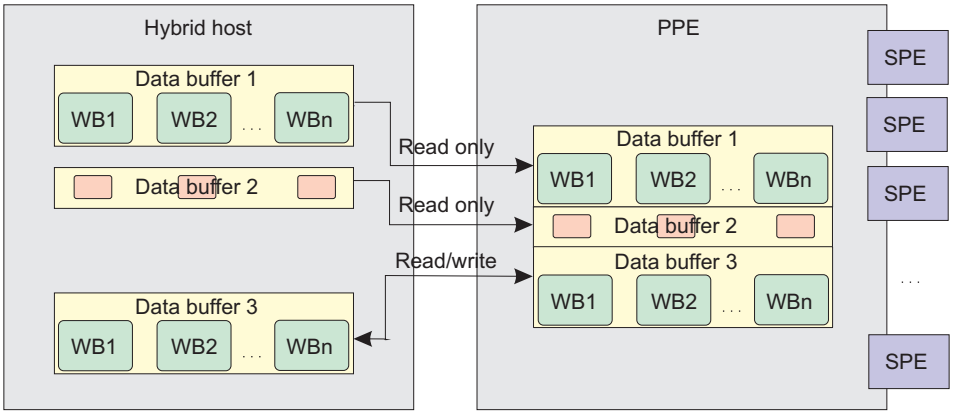

Figure 7. How data sets are used

You set up a data set independently of tasks or work blocks using the `alf_dataset_create`, and `alf_dataset_buffer_add` functions. Before enqueuing the first work block, you must associate the data set to one or more tasks using the `alf_task_dataset_associate` function. As work blocks are enqueued, they are checked against the associated data set to ensure they reside within one of the buffers. Finally after finishing with the data set, you destroy it by using the `alf_dataset_destroy` function.

A data set can have a set of data buffers associated with it. A data buffer can be identified as read-only, write-only, or read and write. You can add as many data buffers to the data set as needed. Different ALF implementations can choose to limit the number of data buffers in a specific data set. Refer to the implementation documentation for restriction information about the number of data buffers in a data set. However, after a data set has been associated with a task, you cannot add additional data buffers to the data set.

A task can optionally be associated with one and only one data set. Work blocks within this task refer to data within the data set for input, output, and in-out buffers. References to work block input and output data which is outside of the data set result in an error. The task context buffer and work block parameter buffer do not need to reside within the data set and are not checked against it.

Multiple tasks can share the same data set. It is your responsibility to make sure that the data in the data set is used correctly. If two tasks with no dependency on each other use the same data from the same data set, ALF cannot guarantee the consistency of the data. For tasks with a dependency on each other and which use the same data set, the data set gets updated in the order in which the tasks are run.

Although for host data partitioning you may create and use data sets, it is recommended that you do use data sets. For accelerator data partitioning you must create and use data sets.

For an example of how to use data sets, see "Data set example" on page 123.

Error handling

ALF supports limited capability to handle runtime errors. Upon encountering an error, the ALF runtime tries to free up resources, then exits by default. To allow the accelerated library developers to handle errors in a more graceful manner, you can register a callback error handler function to the ALF runtime. Depending on the type of error, the error handler function can direct the ALF runtime to retry the current operation, stop the current operation, or shut down. These are controlled by the return values of the callback error handler function.

When several errors happen in a short time or at the same time, the ALF runtime attempts to invoke the error handler in sequential order.

Possible runtime errors include the following:

- Compute task runtime errors such as bus error, undefined computing kernel function names, invalid task execution images, memory allocation issues, dead locks, and others
- Detectable internal data structure corruption errors, which might be caused by improper data transfers or access boundary issues
- Application detectable/catchable errors

Standard error codes on supported platforms are used for return values when an error occurs. For this implementation, the standard C/C++ header file, `errno.h`, is used. See Appendix D, "Error codes and descriptions," on page 137 and also the API definitions in Chapter 14, "ALF API overview," on page 51 for a list of possible error codes.

Part 2. Programming with ALF

This section describes the following ALF programming topics:

For information configuration information including how to switch compilers, see the `alf/README_alf_samples` file.

Chapter 5. Basic structure of an ALF application

The basic structure of an ALF application is shown in Figure 8. The process on the host is as follows:

1. Initialize the ALF runtime.
2. Create a compute task.
3. After the task is created, you start to add work blocks to the work queue of the task.
4. Wait for the task to complete and shut down the ALF runtime to release the allocated resources.

The process on the accelerator is as follows:

1. After an instance of the task is spawned, it waits for pending work blocks to be added to the work queue.
2. The `alf_accel_comp_kernel` function is called for each work block.
3. If the task has been created with a task descriptor with `ALF_TASK_DESC_PARTITION_ON_ACCEL` set to 1, then the `alf_accel_input_dtl_prepare` function is called before the invocation of the compute kernel and the `alf_accel_output_dtl_prepare` function is called after the compute kernel exits.

For examples of ALF applications including some source code samples, see Appendix A, “Examples,” on page 109.

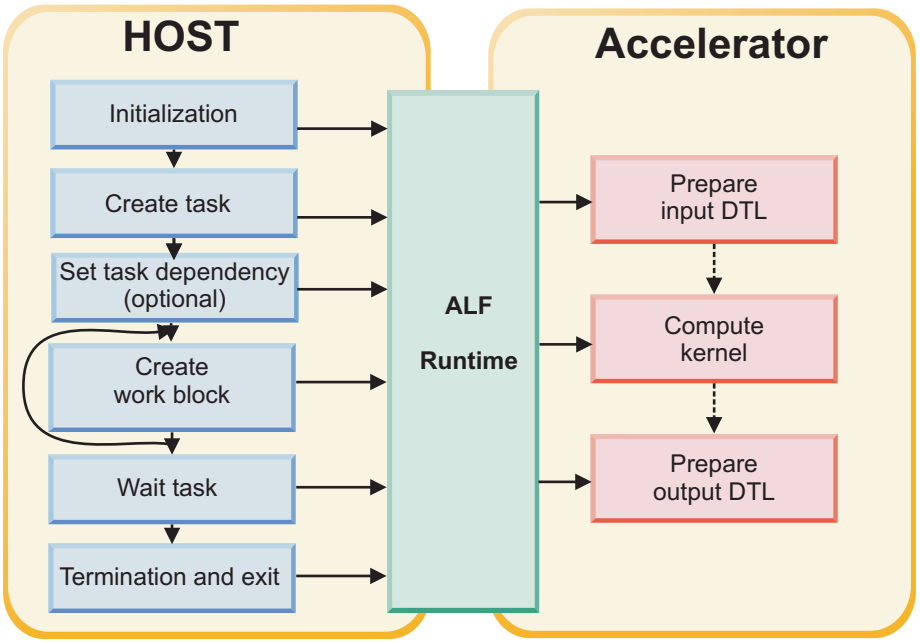

Figure 8. ALF application structure and process flow

Chapter 6. ALF host application and data transfer lists

One important decision to make is whether to use accelerator data transfer list generation.

See Figure 8 on page 27 for the flow diagram of ALF applications on the host. When there are a large number of accelerators used for one compute task and if the data transfer list is complex, the host might not be able to generate work blocks as fast as the accelerators can process them. In this case, you can supply the data needed for data transfer list generation in the parameters of the work block and use the accelerators to generate the data transfer lists based on these parameters. You can also start a new task to be queued while another task is running. You can then prepare the work blocks for the new task before the task runs.

Chapter 7. When to use the overlapped I/O buffer

An overlapped I/O buffer is designed to maximize the memory usage on accelerators. This is particularly useful when there is limited accelerator memory and input and output data. For each task instance, the ALF runtime provides an optional overlapped I/O buffer. This buffer is accessible from the user-defined computational kernel as well as the `input_dtl_prepare` and `output_dtl_prepare` functions. For each overlapped I/O buffer, you can dynamically define three types of buffer area for each work block:

- `ALF_BUF_OVL_IN`: Data in the host memory is copied to this section of the overlapped I/O buffer before the computational kernel is called
- `ALF_BUF_OVL_OUT`: Data in this buffer area of the overlapped I/O buffer is written back to the host memory after the computational kernel is called
- `ALF_BUF_OVL_INOUT`: Data in the host memory is copied to this buffer area before the computational kernel is called and is written back to the same host memory location after the computational kernel is called

For examples of how to use the overlapped I/O buffer, see "Overlapped I/O buffer example" on page 119.

Points to consider when using the overlapped I/O buffer

When you use overlapped I/O buffer, you need to make sure that the input data area defined by `ALF_BUF_OVL_IN` and `ALF_BUF_OVL_INOUT` do not overlap each other. The ALF runtime does not guarantee the order in which the input data is pulled into accelerator memory, so the input data can become corrupted if these two areas are overlapped. Figure 9 shows a corrupted overlapped I/O buffer.

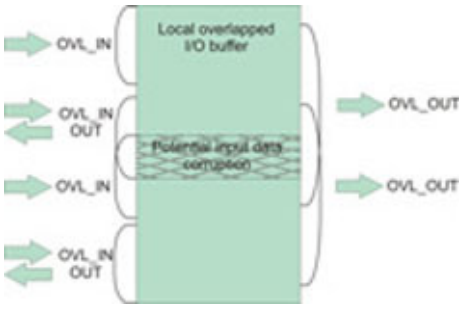

Figure 9. Corrupted overlapped I/O buffer

If you choose to partition data on the accelerator, you need to generate the data transfer lists for the input buffer, the overlapped input buffer, and the overlapped I/O buffer in the user-provided `alf_accel_input_dtl_prepare` function and generate the data transfer lists for both the output buffer and the overlapped output buffer in the user-provided `alf_accel_output_dtl_prepare` function.

Chapter 8. What to consider for data layout design

Efficient data partitioning and data layout design is the key to a well-performed ALF application. Improper data partitioning and data layout design either prevents ALF from being applicable or results in degraded performance. Data partition and layout is closely coupled with compute kernel design and implementation, and hey should be considered simultaneously. You should consider the following for your data layout and partition design:

- Use the correct size for the data partitioned for each work block. Often the local memory of the accelerator is limited. Performance can degrade if the partitioned data cannot fit into the available memory.
- Minimize the amount of data movement. A large amount of data movement can cause performance loss in applications. Improve performance by avoiding unnecessary data movements.
- Simplify data movement patterns. Although the data transfer list feature of ALF enables flexible data gathering and scattering patterns, it is better to keep the data movement patterns as simple as possible. Some good examples are sequential access and using contiguous movements instead of small discrete movements.
- Avoid data reorganization. Data reorganization requires extra work. It is better to organize data in a way that suits the usage pattern of the algorithm than to write extra code to reorganize the data when it is used.

Chapter 9. Double buffering on ALF

When transferring data in parallel with the computation, double buffering can reduce the time lost to data transfer by overlapping it with the computation time. The ALF runtime implementation on Cell BE™ architecture supports three different kinds of double buffering schemes.

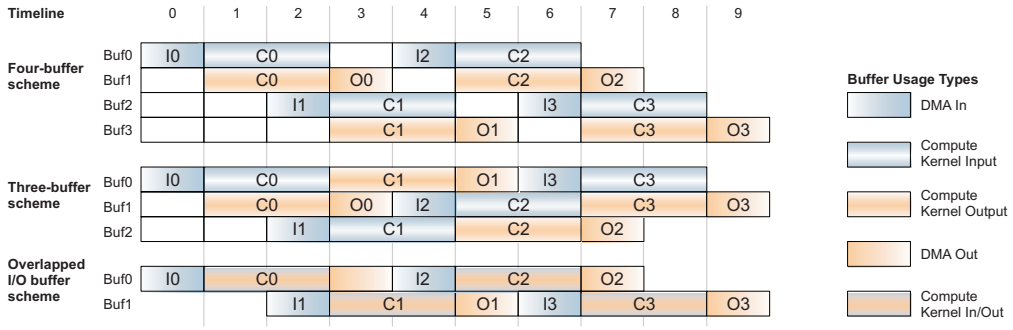

Figure 10. ALF double buffering

See Figure 10 for an illustration of how double buffering works inside ALF. The ALF runtime evaluates each work block and decides which buffering scheme is most efficient. At each decision point, if the conditions are met, that buffering scheme is used. The ALF runtime first checks if the work block uses the overlapped I/O buffer. If the overlapped I/O buffer is not used, the ALF runtime next checks the conditions for the four-buffer scheme, then the conditions of the three-buffer scheme. If the conditions for neither scheme are met, the ALF runtime does not use double buffering. If the work block uses the overlapped I/O buffer, the ALF runtime first checks the conditions for the overlapped I/O buffer scheme, and if those conditions are not met, double buffering is not used.

These examples use the following assumptions:

1. All SPUs have 256 KB of local memory.
2. 16 KB of memory is used for code and runtime data including stack, the task context buffer, and the data transfer list. This leaves 240 KB of local storage for the work block buffers.
3. Transferring data in or out of accelerator memory takes one unit of time and each computation takes two units of time.
4. The input buffer size of the work block is represented as `in_size`, the output buffer size as `out_size`, and the overlapped I/O buffer size as `overlap_size`.
5. There are three computations to be done on three inputs, which produces three outputs.

Buffer schemes

The conditions and decision tree are further explained in the examples below.

- **Four-buffer scheme**: In the four-buffer scheme, two buffers are dedicated for input data and two buffers are dedicated for output data. This buffer use is shown in the Four-buffer scheme section of Figure 10.

  - **Conditions satisfied**: The ALF runtime chooses the four-buffer scheme if the work block does not use the overlapped I/O buffer and the buffer sizes satisfy the following condition: 2*(in_size + out_size) <= 240 KB.
  - **Conditions not satisfied**: If the buffer sizes do not satisfy the four-buffer scheme condition, the ALF runtime will check if the buffer sizes satisfy the conditions of the three-buffer scheme.
- **Three-buffer scheme**: In the three-buffer scheme, the buffer is divided into three equally sized buffers of the size max(in_size, out_size). The buffers in this scheme are used for both input and output as shown in the Three-buffer scheme section of Figure 10 on page 35. This scheme requires the output data movement of the previous result to be finished before the input data movement of the next work block starts, so the DMA operations must be done in order. The advantage of this approach is that for a specific work block, if the input and output buffer are almost the same size, the total effective buffer size can be 2*240/3 = 160 KB.
  - **Conditions satisfied**: The ALF runtime chooses the three-buffer scheme if the work block does not use the overlapped I/O buffer and the buffer sizes satisfy the following condition: 3*max(in_size, out_size) <= 240 KB.
  - **Conditions not satisfied**: If the conditions are not satisfied, the single-buffer scheme is used.
- **Overlapped I/O buffer scheme**: In the overlapped I/O buffer scheme, two contiguous buffers are allocated as shown in the Overlapped I/O buffer scheme section of Figure 10 on page 35. The overlapped I/O buffer scheme requires the output data movement of the previous result to be finished before the input data movement of the next work block starts.
  - **Conditions satisfied**: The ALF runtime chooses the overlapped I/O buffer scheme if the work block uses the overlapped I/O buffer and the buffer sizes satisfy the following condition: 2*(in_size + overlap_size + out_size) <= 240 KB.
  - **Conditions not satisfied**: If the conditions are not satisfied, the single-buffer scheme is used.
- **Single-buffer scheme**: If none of the cases outlined above can be satisfied, double buffering is not used, but performance might not be optimal.

When creating buffers and data partitions, remember the conditions of these buffering schemes. If your buffer sizes can meet the conditions required for double buffering, it can result in a performance gain, but double buffering does not double the performances in all cases. When the time periods required by data movements and computation are significantly different, the problem becomes either I/O-bound or computing-bound. In this case, enlarging the buffers to allow more data for a single computation might improve the performance even with a single buffer.

Chapter 10. Performance and debug trace

The Performance Debugging Tool (PDT) provides trace data necessary to debug functional and performance problems for applications using the ALF library. Versions of the ALF libraries built with PDT trace hooks enabled are delivered with SDK 3.0.

Installing the PDT

The libraries with the trace hooks enabled are packaged in separate "`-trace`" named packages. The trace enabled libraries install to a subdirectory named trace in the library install directories. These packages and the PDT are included in the SDK 3.0 package but may not be installed by default.

Refer to the *PDT User's Guide* for instructions about how to install PDT, and how to set the correct environment variables to cause trace events to be generated. ALF ships example configuration files that list all of the ALF groups and events, and allow you to turn selected ones off as desired. They are located in the `/usr/share/pdt/example` directory.

Chapter 11. Trace control

When a PDT-enabled application starts, PDT reads its configuration from a file. The PDT configuration for DaCS is separate from the configuration for your job.

Environment variable

PDT supports an environment variable (`PDT_CONFIG_FILE`) that allows you to specify the relative or full path to a configuration file.

ALF ships an example configuration file that lists all of the ALF groups and events, and allows the user to turn selected ones off as desired. This is shipped as `/usr/share/pdt/config/pdt_alf_config_cell.xml`

.

Part 3. Programming ALF for Hybrid-x86

This section describes information specific to programming ALF for Hybrid-x86

It describes the following:

- Chapter 12, "Optimizing ALF," on page 43
- Chapter 13, "Platform-specific constraints for the ALF implementation on the Hybrid architecture," on page 45

For installation information, refer to the *SDK for Multicore Acceleration Version 3.0 Installation Guide*.

Chapter 12. Optimizing ALF

This section describes how to optimize your ALF applications. It covers the following topics:

- "Using accelerator data partitioning"
- "Using multi-use work blocks"
- "Using data sets"

Using accelerator data partitioning

If the application operates in an environment where the host has many accelerators to manage and the data partition schemes are particularly complex, it is generally more efficient for the application to partition the data and generate the data transfer lists on the accelerators instead on the host.

For more information about how to use this feature, refer to "Accelerator data partitioning" on page 17.

Using multi-use work blocks

If there are many instances of the task running on the accelerators and the amount of computation per work block is small, the ALF runtime can become overwhelmed with moving work blocks and associated data in and out of accelerator memory. In this case, multi-use work blocks can be used in conjunction with accelerator data partitioning to further improve performance for an ALF application.

For an example of how to use multi-use work blocks, refer to "Implementation 2: Making use of multi-use work blocks together with task context or work block parameter/context buffers" on page 118.

Using data sets

The data set is the primary mechanism for optimizing an ALF application on a hybrid system. Using the data set improves ALF application performance on the hybrid environment significantly. For an overview about data sets, see "Data set" on page 22.

The ALF on Hybrid implementation uses the data set to speed up data read/write access time by migrating the data set closer to the accelerators. That is, when the task is ready and is scheduled for execution on a specific PowerPC Processing Element (PPE), the associated data set is transferred from the Hybrid host to the PPE. This makes the data set available for the accelerator's data access. All read-only and read-write buffers are transferred then. Similarly, when you issue the `alf_task_wait` function, and the task completes, the data set is transferred from the PPE to the Hybrid host. This makes the data set available for the host's data access. All the write-only and read-write buffers are transferred then.

Refer to "Data set example" on page 123 for an example of how to use data sets.

Chapter 13. Platform-specific constraints for the ALF implementation on the Hybrid architecture

This section describes constraints that apply when you program ALF for Cell BE.

Data set constraints

The ALF on Hybrid data set implementation is limited to eight (8) buffers.

Further constraints

Because the ALF for Hybrid implementation is based upon the ALF for Cell BE implementation, the following constraints are applicable to both the Cell BE and Hybrid implementations. The explanations and examples used sometimes only refer to the Cell BE implementation, but are also applicable to both implementations.

Local memory constraints

The size of local memory on the accelerator is 256 KB and is shared by code and data. Memory is not virtualized and is not protected. See Figure 11 on page 46 for a typical memory map of an SPU program. There is a runtime stack above the global data memory section. The stack grows from the higher address to the lower address until it reaches the global data section. Due to the limitation of programming languages and compiler and linker tools, you cannot predict the maximum stack usage when you develop the application and when the application is loaded. If the stack requires more memory that what was allocated you do not get a stack overflow exception (unless this was enabled by the compiler at build time) you get undefined results such as bus error or illegal instruction. When there is a stack overflow, the SPU application is shut down and a message is sent to the PPE.

ALF allocates the work block buffers directly from the memory region above the runtime stack, as shown in Figure 12 on page 46. This is implemented by moving the stack pointer (or equivalently by pushing a large amount of data into the stack). To ALF, the larger the buffer is, the better it can optimize the performance of a task by using techniques like double buffering. It is better to let ALF allocate as much memory as possible from the runtime stack. If the stack size is too small at runtime, a stack overflow occurs and it causes unexpected exceptions such as incorrect results or a bus error.

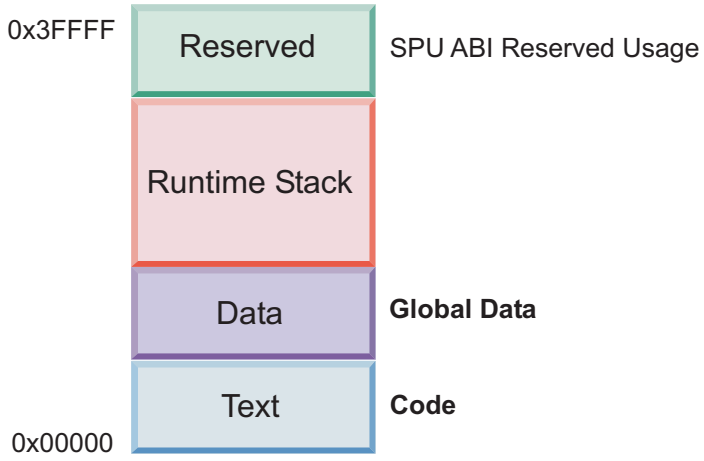

Figure 11. SPU local memory map of a common Cell BE application

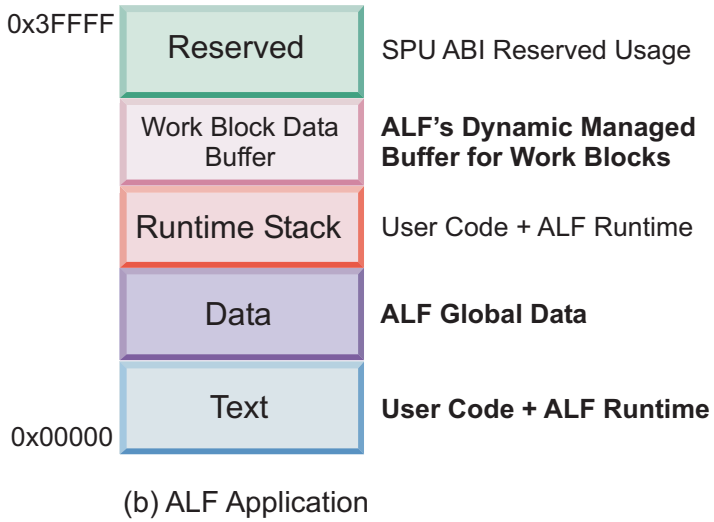

Figure 12. SPU local memory map of an ALF application

Data transfer list limitations

Data transfer information is used to describe the five types data movement operations for one work block as defined by `ALF_BUF_TYPE_T`. The ALF implementation on Cell BE has the following internal constraints:

1. Data transfer information for a single work block can consist of up to eight data transfer lists for each type of transfer as defined by `ALF_BUF_TYPE_T`. For programmers the limitation is that `alf_wb_dtl_begine` can only be called no more than eight times for each kind of `ALF_BUF_TYPE_T` for each work block. An `ALF_ERR_NOBUFS` is returned in this case. Due to limitation items 2, 3 and 4 in this list, it is possible that the limitation can be reached without explicitly calling `alf_wb_dtl_begin` by eight times.
2. Each data transfer list consists of up to 2048 data transfer entries. The `alf_wb_dtl_entry_add` call automatically creates a new data transfer list of the same type when this limitation is reached. Limitation item 1 in this list still applies in this case.

3. Each entry can describe up to 16 KB of data transfer between the contiguous area in host memory and accelerator memory. The `alf_wb_dtl_entry_add` call automatically breaks an entry larger than 16 KB to multiple entries. Limitation items 1 and 2 in this list still apply in this case.
4. All of the entries within the same data transfer list share the same high 32 bits effective address. This means that when a data transfer entry goes across 4 GB address boundary, it must be broken up and put into two different data transfer lists. In addition, two succeeding entries use different high 32 bit addresses, they need to be put into two lists. The `alf_wb_dtl_entry_add` call automatically creates a new data transfer list in the above two situations. Limitation items 1, 2 and 3 in this list still apply in this case.
5. The local store area described by each entry within the same data transfer list must be contiguous. You can use the local buffer offset parameter `"offset_to_accel_buf"` to address with in the local buffer when `alf_wb_dtl_begin` is called to create a new list.
6. The transfer size and the low 32 bits of the effective address for each data transfer entry must be 16 bytes aligned. The `alf_wb_dtl_entry_add` call does NOT help you to automatically deal with alignment issues. An `ALF_ERR_INVAL` error is returned if there is an unaligned address. The same limitation also applies to the `offset_to_accel_buf` parameter of `alf_wb_dtl_begin`.

Part 4. API reference

This section covers the following topics:

Chapter 14. ALF API overview

Conventions

ALF and *alf* are the prefixes for the namespace for ALF. For normal function prototypes and data structure declarations, use all lowercase characters with underscores (_) separating the words. For macro definitions, use all uppercase characters with underscores separating the words.

Data type assumptions

`int`	This data type is assumed to be signed by default on both the host and accelerator. The size of this data type is defined by the Application Binary Interface (ABI) of the architecture. However, the minimum size of this data type is 32 bits. The actual size of this data type might differ between the host and the accelerator architectures.
`unsigned int`	This data type is assumed to be the same size as that of int.
`char`	This data type is not assumed to be signed or unsigned. The size of this data structure, however, must be 8 bits.
`long`	This data type is not used in the API definitions because it might not be uniformly defined across platforms.
`void *`	The size of this data type is defined by the ABI of the corresponding architecture and compiler implementation. Note that the actual size of this data type might differ between the host and accelerator architectures.

Platform-dependent auxiliary APIs or data structures

The basic APIs and data structures of ALF are designed with cross-platform portability in mind. Platform-dependent implementation details are not exposed in the core APIs.

Common data structures

The enumeration type `ALF_DATA_TYPE_T` defines the data types for data movement operations between the hosts and the accelerators. The ALF runtime does byte swapping automatically if the endianness of the host and the accelerators are different. To disable endian swapping, you can use the data type `ALF_DATA_BYTE`.

`ALF_DATA_BYTE`	For data types that are independent of byte orders
`ALF_DATA_INT16`	For two bytes signed / unsigned integer types
`ALF_DATA_INT32`	For four bytes signed / unsigned integer types
`ALF_DATA_INT64`	For eight bytes signed / unsigned integer types
`ALF_DATA_FLOAT`	For four bytes float point types
`ALF_DATA_DOUBLE`	For eight bytes float point types
`ALF_DATA_ADDR32`	32-bit address
`ALF_DATA_ADDR64`	64-bit address

ALF_NULL_HANDLE

The constant `ALF_NULL_HANDLE` is used to indicate a non-initialized handle in the ALF runtime environment. All handles should be initialized to this value to avoid ambiguity in code semantics.

ALF runtime APIs that create handles always return results through pointers to handles. After the API call is successful, the original content of the handle is overwritten. Otherwise, the content is kept unchanged. ALF runtime APIs that destroy handles modify the contents of handle pointers and initialize the contents to `ALF_NULL_HANDLE`.

ALF_STRING_TOKEN_ MAX

This constant defines the maximum allowed length of the string tokens in unit of bytes, excluding the trailing zero. These string tokens are used in ALF as identifiers of function names or other purposes. Currently, this value is defined to be 251 bytes.

Chapter 15. Host API

The host API includes the following:

Basic framework API

The following API definitions are the basic framework APIs.

alf_handle_t

This data structure is used as a reference to one instance of the ALF runtime. The data structure is initialized by calling the `alf_init` API call and is destroyed by `alf_exit`.

ALF_ERR_POLICY_T

NAME

ALF_ERR_POLICY_T - Callback function prototype that can be registered to the ALF runtime for customized error handling.

SYNOPSIS

ALF_ERR_POLICY_T(*alf_error_handler_t)(void *p_context_data, int error_type, int error_code, char *error_string)

Parameters

`p_context_data [IN]` A pointer given to the ALF runtime when the error handler is registered. The ALF runtime passes it to the error handler when the error handler is invoked. The error handler can use this pointer to keep its private data.

`error_type [IN]` A system-wide definition of error type codes, including the following:

- `ALF_ERR_FATAL`: Cannot continue, the framework must shut down.
- `ALF_ERR_EXCEPTION`: You can choose to retry or skip the current operation.
- `ALF_ERR_WARNING`: You can choose to continue by ignoring the error.

`error_code [IN]` A type-specific error code.

`error_string [IN]` A C string that holds a printable text string that provides information about the error.

DESCRIPTION

This is a callback function prototype that can be registered to the ALF runtime for customized error handling.

RETURN VALUE

`ALF_ERR_POLICY_RETRY` Indicates that the ALF runtime should retry the operation that caused the error. If a severe error occurs and the ALF runtime cannot retry this operation, it will report an error and shut down.

`ALF_ERR_POLICY_SKIP` Indicates that the ALF runtime should stop the operation that caused the error and continue processing. If the error is severe and the ALF runtime cannot continue, it will report an error and shut down.

`ALF_ERR_POLICY_ABORT` Indicates that the ALF runtime must stop the operations and shut down.

`ALF_ERR_POLICY_IGNORE` Indicates that the ALF runtime will ignore the error and continue. If the error is severe and the ALF runtime cannot continue, it will report an error and shut down.

EXAMPLES

See “alf_register_error_handler” on page 62 for an example of this function.

alf_init

NAME

alf_init - Initializes the ALF runtime.

SYNOPSIS

int alf_init(void* p_sys_config_info, alf_handle_t* p_alf_handle);

Parameters

`p_sys_config_info [IN]`

A platform-dependent configuration information placeholder so that the ALF can get the necessary data for system configuration information.

This parameter should point to an `alf_sys_config_t_Hybrid_t` data structure. This data structure is defined as followed;

```
typedef struct {
  char* library_path;
  unsigned int num_of_ppes;
  char* ppe_image_path;
  char* ppe_image_name;
  char* ppe_image_mode;
 } alf_sys_config_t_Hybrid;
```

Implementation details: This data pointer shall be passed to the PAL layer and is interpreted by the PAL layer.

`p_alf_handle [OUT]`

A pointer to a handle for a data structure that represents the ALF runtime. This buffer is initialized with proper data if the call is successful. Otherwise, the content is not modified.

DESCRIPTION

This function initializes the ALF runtime. It allocates the necessary resources and global data for ALF as well as sets up any platform specific configurations.

RETURN VALUE

>= 0

Successful, the result of the query

less than 0

Errors:

- `ALF_ERR_INVAL`: Invalid input parameter
- `ALF_ERR_NODATA`: Some system configuration data is not available
- `ALF_ERR_NOMEM`: Out of memory or some system resources have been used up
- `ALF_ERR_GENERIC`: Generic internal errors

OPTIONS

Field value

`library_path`

The path to all of the application's computational kernel shared object files. If the pointer is NULL, the `ALF_LIBRARY_PATH` environment variable is checked and if it is defined then it is used. If neither is set, the default "." is used.

num_of_ppes	The number of PPEs to use for application execution. If the value is 0, the ALF_NUM_OF_PPES environment variable is checked and if it is defined then it is used. If neither is set, the default 0 (use all available PPEs) is used.
ppe_image_path	Depending upon the ppe_image_mode (see below), it is either the remote path to the application's PPE daemon or the local path to the application's file list which are to be transferred to the PPE and the first executed. If the pointer is NULL, the ALF_PPE_IMAGE_PATH environment variable is checked and if it is defined then it is used. If neither is set, the default "/opt/cell/sdk/prototype/usr/bin" is used.
ppe_image_name	Depending upon the ppe_image_mode (see below), it is either the name of the application's PPE daemon, or the name of the application's file list. If the pointer is NULL, the ALF_PPE_IMAGE_NAME environment variable is checked and if it is defined then it is used. If neither is set, the default "alf_hybrid_d_ppu64" is used.
ppe_image_mode	The mode to interpret the ppe_image_path and ppe_image_name (see above). It is either the case-insensitive string "remote" or "filelist". If the pointer is NULL, the ALF_PPE_IMAGE_MODE environment variable is checked and if it is defined then it is used. If neither is set, the default "remote" is used.

alf_query_system_info

NAME

alf_query_system_info - Queries basic configuration information.

SYNOPSIS

int alf_query_system_info(alf_handle_t alf_handle, ALF_QUERY_SYS_INFO_T query_info, ALF_ACCEL_TYPE_T accel_type, unsigned int * p_query_result);

Parameters

`alf_handle [IN]` Handle to the ALF runtime.

`query_info [IN]` A query identification that indicates the item to be queried:

- `ALF_QUERY_NUM_ACCEL`: Returns the number of accelerators in the system.
- `ALF_QUERY_HOST_MEM_SIZE`: Returns the memory size of control nodes up to 4T bytes, in units of kilobytes (2^10 bytes). When the size of memory is more than 4T bytes, the total reported memory size is (`ALF_QUERY_HOST_MEM_SIZE_EXT`*4T + `ALF_QUERY_HOST_MEM_SIZE`*1K) bytes. In case of systems where virtual memory is supported, this should be the maximum size of one contiguous memory block that a single user space application could allocate.
- `ALF_QUERY_HOST_MEM_SIZE_EXT`: Returns the memory size of control nodes, in units of 4T bytes (2^42 bytes).
- `ALF_QUERY_ACCEL_MEM_SIZE`: Returns the memory size of accelerator nodes up to 4T bytes, in units of kilo bytes (2^10 bytes) . When the size of memory is more than 4T bytes, the total reported memory size is (`ALF_QUERY_ACCEL_MEM_SIZE_EXT`*4T + `ALF_QUERY_ACCL_MEM_SIZE`*1K) bytes. For systems where virtual memory is supported, this should be the maximum size of one contiguous memory block that a single user space application could allocate.
- `ALF_QUERY_ACCEL_MEM_SIZE_EXT`: Returns the memory size of accelerator nodes, in units of 4T bytes (2^42 bytes).
- `ALF_QUERY_HOST_ADDR_ALIGN`: Returns the basic requirement of memory address alignment on control node side, in exponential of 2. A zero stands for byte aligned address. A 4 is to align by 16 byte boundaries.
- `ALF_QUERY_ACCEL_ADDR_ALIGN`: Returns the basic requirement of memory address alignment on accelerator node side, in exponential of 2. A zero stands for byte aligned address. An 8 is to align by 256 byte boundaries
- `ALF_QUERY_DTL_ADDR_ALIGN`: Returns the address alignment of data transfer list entries, in exponential of 2. A zero stands for byte aligned address. An 8 is to align by 256 byte boundaries.
- `ALF_QUERY_ACCEL_ENDIAN_ORDER`:
  - `ALF_ENDIAN_ORDER_BIG`
  - `ALF_ENDIAN_ORDER_LITTLE`
- `ALF_QUERY_HOST_ENDIAN_ORDER`:
  - `ALF_ENDIAN_ORDER_BIG`
  - `ALF_ENDIAN_ORDER_LITTLE`

`accel_type [IN]` Accelerator type. There is only one accelerator type defined, which is `ALF_ACCEL_TYPE_SPE`

`p_query_result [OUT]` Pointer to a buffer where the return value of the query is saved. If the query fails, the result is undefined. If a NULL pointer is provided, the query value is not returned, but the call returns zero.

DESCRIPTION

This function queries basic configuration information for the specific system on which ALF is running.

RETURN VALUE

0	Successful, the result of query is returned by `p_result` if that pointer is not NULL
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Unsupported query • `ALF_BADF`: Invalid ALF handle • `ALF_ERR_GENERIC`: Generic internal errors

alf_num_instances_set

NAME

alf_num_instances_set - Sets the maximum total number of parallel task instances ALF can have at one time.

SYNOPSIS

int alf_num_instances_set(alf_handle_t alf_handle, unsigned int number_of_instances);

Parameters

`alf_handle [IN]`	A handle to the ALF runtime code.
`number_of_instances [IN]`	Specifies the maximum number of task instances that the caller wants to have. When this parameter is zero, the runtime allocates as many task instances as requested by the application programmer. However, the subsequent `alf_ask_create` call returns an error if ALF cannot accommodate the request.

DESCRIPTION

This function sets the maximum total number of parallel task instances ALF can have at one time. If `number_of_instances` is zero, there is no limit set by the application and ALF returns an error if it cannot accommodate a particular task creation request with a large number of instances.

Note: In SDK 3.0, this function is called once at the beginning after `alf_init` and before any `alf_task_create`. The ability to call this function twice to reset the number of instances is not supported. An `ALF_ERR_PERM` is returned in this situation.

RETURN VALUE

> 0	the actual number of instances provided by the ALF runtime.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid ALF handle • `ALF_ERR_PERM`: The API call is not permitted at the current context • `ALF_ERR_GENERIC`: Generic internal errors

alf_exit

NAME

alf_exit - Shuts down the ALF runtime.

SYNOPSIS

int alf_exit(alf_handle_t alf_handle, ALF_EXIT_POLICY_T policy, int timeout);

Parameters

`alf_handle [IN]`
The ALF handle

`policy [IN]`
Defines the shutdown behavior:

- `ALF_EXIT_POLICY_FORCE`: Performs a shutdown immediately and stops all unfinished tasks if there are any.
- `ALF_EXIT_POLICY_WAIT`: Waits for all tasks to be processed and then shuts down.
- `ALF_EXIT_POLICY_TRY`: Returns with a failure if there are unfinished tasks.

`time_out [IN]`
A timeout value that has the following values:

- > 0 : Wait at most the specified milliseconds before a timeout error happens or a forced shutdown
- = 0 : Shutdown or return without wait
- less than 0 : Waits forever, only valid with `ALF_EXIT_POLICY_WAIT`

DESCRIPTION

This function shuts down the ALF runtime. It frees allocated accelerator resources and stops all running or pending work queues and tasks, depending on the `policy` parameter.

RETURN VALUE

>= 0
The shutdown succeeded. The number of unfinished work blocks is returned.

less than 0
The shutdown failed:

- `ALF_ERR_INVAL`: Invalid input argument
- `ALF_ERR_BADF`: Invalid ALF handle
- `ALF_ERR_PERM`: The API call is not permitted at the current context
- `ALF_ERR_NOSYS`: The required policy is not supported
- `ALF_ERR_TIME`: Timeout
- `ALF_ERR_BUSY`: There are tasks still running
- `ALF_ERR_GENERIC`: Generic internal errors

alf_register_error_handler

NAME

alf_register_error_handler - Registers a global error handler function to the ALF runtime code.

SYNOPSIS

int alf_register_error_handler(alf_handle_t alf_handle, alf_error_handler_t error_handler_function, void *p_context)

Parameters

`alf_handle [IN]`	A handle to the ALF runtime code.
`error_handler_function [IN]`	A pointer to the user-defined error handler function. A NULL value resets the error handler to the ALF default handler.
`p_context [IN]`	A pointer to the user-defined context data for the error handler function. This pointer is passed to the user-defined error handler function when it is invoked.

DESCRIPTION

This function registers a global error handler function to the ALF runtime code. If an error handler has already been registered, the new one replaces it.

RETURN VALUE

0	Successful.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid ALF handle • `ALF_ERR_PERM`: The API call is not permitted at the current context • `ALF_ERR_FAULT`: Invalid buffer or error handler address (only when it is possible to detect the fault) • `ALF_ERR_GENERIC`: Generic internal errors

Compute task API

The following API definitions are the compute task APIs.

alf_task_handle_t

This data structure is a handle to a specific compute task running on the accelerators. It is created by calling the `alf_task_create` function and destroyed by either calling the `alf_task_destroy` function or when the `alf_exit` function is called. Call the `alf_task_wait` function to wait for the task to finish processing all queued work blocks. The `alf_task_wait` API is also an indication to the ALF runtime that no new work blocks will be added to the work queue of the corresponding task in the future.

alf_task_desc_handle_t

This data structure is a handle to a task descriptor. It is used to access and setup task descriptor information. It is created by calling `alf_task_desc_create` and destroyed by calling `alf_task_desc_destroy`.

alf_task_desc_create

NAME

alf_task_desc_create - Creates a task descriptor.

SYNOPSIS

int alf_task_desc_create (alf_handle_t alf_handle, ALF_ACCEL_TYPE_T accel_type, alf_task_desc_handle_t * p_desc_info_handle);

Parameters

`alf_handle`	Handle to the ALF runtime.
`accel_type [IN]`	The type of accelerator that tasks created from this descriptor are expected to run on.
`p_task_desc_handle [OUT]`	Returns a handle to the created task description. The content of the pointer is not modified if the call fails.

DESCRIPTION

This function creates a task descriptor. The data structure is returned through the pointer to its handle. The created data structure contains all the information relevant for a compute task.

RETURN VALUE

0	Successful
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid ALF handle • `ALF_ERR_NOMEM`: Out of memory or system resource • `ALF_ERR_PERM`: The API call is not permitted at the current context • `ALF_ERR_GENERIC`: Generic internal errors

alf_task_desc_destroy

NAME

alf_task_desc_destroy - Destroys the specified task descriptor and frees up the resources associated with this task descriptor.

SYNOPSIS

int alf_task_desc_destroy (alf_task_desc_handle_t task_desc_handle);

Parameters

`task_desc_handle [IN/OUT]` Handle to a task descriptor. This data structure is destroyed when it returns from this call.

DESCRIPTION

This function destroys the specified task descriptor and frees up the resources associated with this task descriptor. A task descriptor cannot be destroyed if it is being used by a task. An attempt to destroy an occupied task descriptor results in an error.

RETURN VALUE

0	Successful
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_BADF`: Invalid task descriptor handle. • `ALF_ERR_BUSY`: This task descriptor is being used. You must destroy all tasks using this descriptor before you can destroy the descriptor. • `ALF_ERR_PERM`: The API call is not permitted at the current context. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_task_desc_ctx_entry_add

NAME

alf_task_desc_ctx_entry_add - Adds a description of one entry in the task context associated with this task descriptor.

SYNOPSIS

int alf_task_desc_ctx_entry_add (alf_task_desc_handle_t task_desc_handle, ALF_DATA_TYPE_T data_type, unsigned int size);

Parameters

`task_desc_handle [IN]`	Handle to the task descriptor structure
`data_type [IN]`	Data type of data in the entry
`size [IN]`	Number of elements of type `data_type`

DESCRIPTION

This function adds a description of one entry in the task context associated with this task descriptor.

RETURN VALUE

0	Successful
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid task descriptor handle • `ALF_ERR_NOSYS`: The `ALF_DATA_TYPE_T` provided is not supported. • `ALF_ERR_PERM`: The API call is not permitted at the current context • `ALF_ERR_NOBUFS`: The requested entry has exceeded the maximum buffer size • `ALF_ERR_GENERIC`: Generic internal errors

alf_task_desc_set_int32

NAME

alf_task_desc_set_int32 - Sets the value for a specific integer field of the task descriptor.

SYNOPSIS

int alf_task_desc_set_int32 (alf_task_desc_handle_t task_desc_handle, ALF_TASK_DESC_FIELD_T field, unsigned int value);

Parameters

`task_desc_handle [IN/OUT]` Handle to the task descriptor structure

`field [IN]` The field to be set. Possible inputs are

- `ALF_TASK_DESC_WB_PARM_CTX_BUF_SIZE`: size of the work block parameter buffer
- `ALF_TASK_DESC_WB_IN_BUF_SIZE`: size of the work block input buffer
- `ALF_TASK_DESC_WB_OUT_BUF_SIZE`: size of the work block output buffer
- `ALF_TASK_DESC_WB_INOUT_BUF_SIZE`: size of the work block overlapped input/output buffer
- `ALF_TASK_DESC_NUM_DTL_ENTRIES`: maximum number of entries for the data transfer list
- `ALF_TASK_DESC_TSK_CTX_SIZE`: size of the task context buffer
- `ALF_TASK_DESC_PARTITION_ON_ACCEL`: specifies whether the accelerator functions (`alf_accel_input_dtl_prepare` and `alf_accel_output_dtl_prepare`) are invoked to generate data transfer lists for input and output data.
- `ALF_TASK_DESC_MAX_STACK_SIZE`:

`value [IN]` New value of the specified field

DESCRIPTION

This function sets the value for a specific integer field of the task descriptor.

RETURN VALUE

0 Successful

less than 0 Errors occurred:

- `ALF_ERR_INVAL`: Invalid input argument
- `ALF_ERR_BADF`: Invalid task descriptor handle
- `ALF_ERR_NOSYS`: The `ALF_TASK_DESC_FIELD` provided is not supported.
- `ALF_ERR_PERM`: The API call is not permitted at the current context
- `ALF_ERR_RANGE`: The specified value is out of the allowed range
- `ALF_ERR_GENERIC`: Generic internal errors

alf_task_desc_set_int64

NAME

alf_task_desc_set_int64 - Sets the value for a specific long integer field of the task descriptor structure.

SYNOPSIS

int alf_task_desc_set_int64(alf_task_desc_handle_t task_desc_handle, ALF_TASK_DESC_FIELD_T field, unsigned long long value);

Parameters

`task_desc_handle [IN/OUT]` Handle to the task descriptor structure

`field [IN]` The field to be set. Possible inputs are

- `ALF_TASK_DESC_ACCEL_LIBRARY_REF_L`: Specify the name of the library that the accelerator image is contained in.
- `ALF_TASK_DESC_ACCEL_IMAGE_REF_L` : Specify the name of the accelerator image that is contained in the library.
- `ALF_TASK_DESC_ACCEL_KERNEL_REF_L`: Specify the name of the computational kernel function, this usually is a string constant that the accelerator runtime could use to find the correspondent function.
- `ALF_TASK_DESC_ACCEL_INPUT_DTL_REF_L`: Specify the name of the input list prepare function, this usually is a string constant that the accelerator runtime could use to find the correspondent function.
- `ALF_TASK_DESC_ACCEL_OUTUT_DTL_REF_L`: Specify the name of the output list prepare function, this usually is a string constant that the accelerator runtime could use to find the correspondent function
- `ALF_TASK_DESC_ACCEL_CTX_SETUP_REF_L`: Specify the name of the context setup function, this usually is a string constant that the accelerator runtime could use to find the correspondent function.
- `ALF_TASK_DESC_ACCEL_CTX_MERGE_REF_L`: Specify the name of the context merge function, this usually a string constant that the accelerator runtime could use to find the correspondent function.

`value [IN]` New value of the specified field

DESCRIPTION

This function sets the value for a specific long integer field of the task descriptor structure. All string constants must have a maximum number of `ALF_STRING_TOKEN_MAX` size.

RETURN VALUE

0 Successful

less than 0	Errors occurred: • ALF_ERR_INVAL: Invalid input argument • ALF_ERR_BADF: Invalid task descriptor handle • ALF_ERR_NOSYS: The ALF_TASK_DESC_FIELD provided is not supported. • ALF_ERR_PERM: The API call is not permitted at the current context • ALF_ERR_RANGE: The specified value is out of the allowed range • ALF_ERR_GENERIC: Generic internal errors

alf_task_create

NAME

alf_task_create - Creates a task and allows you to add work blocks to the work queue of the task.

SYNOPSIS

int alf_task_create(alf_task_desc_handle_t task_desc_handle, void* p_task_context_data, unsigned int num_instances, unsigned int tsk_attr, unsigned int wb_dist_size, alf_task_handle_t *p_task_handle);

Parameters

`task_desc_handle [IN]`	Handle to a `task_desc` structure.
`p_task_context_data [IN]`	Pointer to the task context data for this task. The structure and size for the task context have been defined through `alf_task_desc_add_task_ctx_entry`. If there is no `task_context`, a NULL pointer can be provided.
`num_instances [IN]`	Number of instances of the task, only used when `ALF_TASK_ATTR_SCHED_FIXED` is provided.
`tsk_attr [IN]`	Attribute for a task. This value can be set to a bit-wise OR to one of the following: • `ALF_TASK_ATTR_SCHED_FIXED`: The task must be scheduled on the specified number of accelerators. By default, a task can be scheduled on any number of accelerators and the number of accelerators can be adjusted at anytime during the execution of the task. • `ALF_TASK_ATTR_WB_CYCLIC`: the work blocks for this task are distributed to the accelerators in a cyclic order as specified by `num_accelerators`. By default, the work blocks distribution order is determined by the ALF runtime. This option must be used combined with `ALF_TASK_ATTR_SCHED_FIXED`.
`wb_dist_size [IN]`	The specified block distribution bundle size in number of work blocks per distribution unit. A 0 (zero) value is treated as 1 (one).
`p_task_handle [OUT]`	Returns a handle to the created task. The content of the pointer is not modified if the call returns failure.

DESCRIPTION

This function creates a task and allows you to enqueue work blocks to the task. The task remains in a pending status until the following condition is met: All dependencies are satisfied and either at least one work block is added or the task is finalized by calling `alf_task_finalize`.

When the condition is met, the task becomes ready to run. However, when the task actually starts to run, depends on the available accelerator resources and the scheduling of ALF runtime. Multiple independent tasks can also run concurrently if there are enough accelerator resources. When the task starts to run, it keeps running until at least one of the following two conditions is met:

- The task has been finalized by calling `alf_task_finalize` and all the enqueued work blocks are processed and the task context has been merged and written back;
- `alf_task_destroy` is called to explicitly destroy the task.

Note: A finalized task without any work block enqueued is never be actually loaded and run. The runtime considers this task as completed immediately after the dependencies are satisfied.

RETURN VALUE

0	Successful
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid ALF handle • `ALF_ERR_NOMEM`: Out of memory or system resource • `ALF_ERR_PERM`: The API call is not permitted at the current context • `ALF_ERR_NOEXEC`: Invalid task image format or description information • `ALF_ERR_2BIG`: Memory requirement for the task exceeds maximum range • `ALF_ERR_NOSYS`: The required task attribute is not supported • `ALF_ERR_BADR`: The requested number of accelerator resources is not available • `ALF_ERR_GENERIC`: Generic internal errors

alf_task_finalize

NAME

alf_task_finalize - Finalizes the work block queue of the specified task.

SYNOPSIS

int alf_task_finalize (alf_task_handle_t task_handle)

Parameters

`task_handle [IN]`	The task handle that is returned by the `alf_create_task` API

DESCRIPTION

This function finalizes the task. After the task has been finalized, future calls to `alf_wb_create` and `alf_task_depends_on` and `alf_task_event_handler_register` return errors.

Note: Task finalization is a compulsory condition for a task to run and complete normally.

RETURN VALUE

less than 0	Errors occurred: • `ALF_ERR_BADF`: Invalid task descriptor handle. • `ALF_ERR_SRCH`: Already finalized task handle. • `ALF_ERR_PERM`: The API call is not permitted at the current context. For example, some created work block handles are not enqueued. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_task_wait

NAME

alf_task_wait - Waits for the specified task to finish processing all work blocks on all the scheduled accelerators.

SYNOPSIS

int alf_task_wait(alf_task_handle_t task_handle, int time_out);

Parameters

`task_handle [IN]` A task handle that is returned by the `alf_create_task` API.

`time_out [IN]` A timeout input with the following options for values:

- > 0: Waits for up to the number of milliseconds specified before a timeout error occurs.
- less than 0: Waits until all of the accelerators finish processing.
- 0: Returns immediately.

DESCRIPTION

This function waits for the specified task to finish processing all work blocks on all the scheduled accelerators. The task must be finalized (`alf_task_finalize` must be called) before this function is called. Otherwise, an `ALF_ERR_PERM` is returned. Data referenced by the task's work blocks can only be used safely after this function returns. If the host application updates the data buffers referenced by work blocks or the task context buffer while the task is running, the result can be undetermined. If you need to update the buffer contents, the only safe point is before the `ALF_TASK_EVENT_READY` task event is handled by the task event handler registered by `alf_task_event_handler_register`.

RETURN VALUE

0 All of the accelerators finished the job.

less than 0 Errors occurred:

- `ALF_ERR_INVAL`: Invalid input argument.
- `ALF_ERR_BADF`: Invalid task handle.
- `ALF_ERR_NODATA`: The task is (during wait) or was (before wait) destroyed explicitly.
- `ALF_ERR_TIME`: Timeout.
- `ALF_ERR_PERM`: The API is not permitted at the current context. For example, the task is not finalized.
- `ALF_ERR_GENERIC`: Generic internal errors.

alf_task_query

NAME

alf_task_query - Queries the current status of a task.

SYNOPSIS

int alf_task_query(alf_task_handle_t task_handle, unsigned int *p_unfinished_wbs, unsigned int *p_total_wbs);

Parameters

`task_handle [IN]`	The task handle to be checked.
`p_unfinished_wbs [OUT]`	A pointer to an integer buffer where the number of unfinished work blocks of this task is returned. When a NULL pointer is given, the return value is ignored. On error, a returned value is not defined.
`p_total_wbs [OUT]`	A pointer to an integer buffer where the total number of submitted work blocks of this task is returned. When a NULL pointer is given, the return value is ignored. On error, a returned value is not defined.

DESCRIPTION

This function queries the current status of a task.

RETURN VALUE

> 1	The task is pending or ready to run.
1	The task is currently running.
0	The task finished normally.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_BADF`: Invalid task handle. • `ALF_ERR_NODATA`: The task was explicitly destroyed. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_task_destroy

NAME

alf_task_destroy - Destroys the specified task.

SYNOPSIS

int alf_task_destroy(alf_task_handle_t* p_task_handle)

Parameters

`task_handle [IN]`	The pointer to a task handle that is returned by the `alf_create_task` API.

DESCRIPTION

This function explicitly destroys the specified task if it is in pending or running state. If there are work blocks that are still not processed, this routine stops the execution of those work blocks. If a task is running when this API is invoked, the task is cancelled before the API returns. Resources associated with this task are recycled by the runtime either synchronously or asynchronously, depending on the runtime implementation. This API does nothing on an already completed task. If a task is destroyed explicitly, all tasks that depend on this task directly or indirectly are destroyed. Because ALF frees task resources automatically, it is not necessary to call this API to free up resources after a task has been run to complete normally. The API should only be used to explicitly end a task when you need to.

RETURN VALUE

0	Success
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_BADF`: Invalid task handle. • `ALF_ERR_PERM`: The API call is not permitted at current context. • `ALF_ERR_BUSY`: Resource busy. • `ALF_ERR_SRCH`: Already destroyed task handle. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_task_depends_on

NAME

alf_task_depends_on - Describes a relationship between two tasks.

SYNOPSIS

int alf_task_depends_on (alf_task_handle_t task_handle_dependent, alf_task_handle_t task_handle);

Parameters

`task_handle_dependent [IN]`	The handle to the dependent task
`task_handle [IN]`	The handle to a task

DESCRIPTION

This function describes a relationship between two tasks. The task specified by `task_handle_dependent` cannot be scheduled to run until the task specified by `task_handle` has run to finish normally. When this API is called, `task_handle` must not be an explicitly destroyed task. An error is reported if it is the case. If the task associated with `task_handle` is destroyed before normal completion, the `task_handle_dependent` is also destroyed because its dependency can no longer be satisfied.

If task A depends on task B, a call to `alf_task_wait` (A_handle) effectively enforces a wait on task B as well. A duplicate dependency is handled silently and not treated as an error.

Note: This function can only be called before any work blocks are enqueued to the `task_handle_dependent` and before the `task_handle_dependent` is finalized. For the `task_handle`, these constraint is not applicable.

Whenever a situation occurs that is not permitted, the function returns `ALF_ERR_PERM`.

RETURN VALUE

0	Success
less than 0	Errors occurred: • `ALF_ERR_BADF`: Invalid task handle. • `ALF_ERR_PERM`: The API call is not permitted at the current context. For example, the dependency cannot be set because of the current state of the task. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_task_event_handler_register

NAME

alf_task_event_handler_register - Allows you to register and unregister an event handler for a specific task.

SYNOPSIS

int alf_task_event_handler_register (alf_task_handle_t task_handle, int (*task_event_handler)(alf_task_handle_t task_handle, ALF_TASK_EVENT_TYPE_T event, void* p_data), void* p_data, unsigned int data_size, unsigned int event_mask);

Parameters

`task_handle [IN]` The handle to a task.

`task_event_handler [IN]` Pointer of the event handler function for the specified task. A NULL value indicates the current event handler is to be unregistered.

`p_context [IN]` A pointer to a context buffer that is copied to another buffer managed by the ALF runtime. The pointer to this buffer is passed to the event handler. The content of the context buffer is copied by value only. A NULL value indicates no context buffer.

`context_size [IN]` The size of the context buffer in bytes. Zero indicates no context buffer.

`event_mask [IN]` A bitwise OR of `ALF_TASK_EVENT_TYPE_T` values. `ALF_TASK_EVENT_TYPE_T` is defined as follows:

- `ALF_TASK_EVENT_FINALIZED`: This task has been finalized. No additional work block can be added to this task. The registered event handler is invoked right before `alf_task_finalize` returns.
- `ALF_TASK_EVENT_READY`: This task has been scheduled for execution. The registered event handler is invoked as soon as the ALF runtime determines that all dependencies have been satisfied for this specific task and can schedule this task for execution as soon as this event handler returns.
- `ALF_TASK_EVENT_FINISHED`: All work blocks in this task have been processed. The registered event handler is invoked as soon as the last work block has been processed and the task context is written back to host memory.
- `ALF_TASK_EVENT_INSTANCE_START`: One new instance of the task is started on an accelerator after the event handler returns.
- `ALF_TASK_EVENT_INSTANCE_END`: One existing instance of the task ends and the task context has been copied out to the original location or has been merged to another current instance of the same task. The event handler is called as soon as the task instance is ended and unloaded from the accelerator.
- `ALF_TASK_EVENT_DESTROY`: The task is destroyed explicitly.

DESCRIPTION

This function allows you to register an event handler for a specified task. This function can only be called before `alf_task_finalize` is invoked. An error is returned if a you try to register an event handler for a task that has been finalized.

If the `task_event_handler` function is NULL, this function unregisters the current event handler. If there is no current event handler, nothing happens.

Note: If the event handler is registered after the task begin to run, some of the events may not be seen.

RETURN VALUE

0	Success
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input handle. • `ALF_ERR_BADF`: Invalid ALF task handle. • `ALF_ERR_PERM`: The API call is not permitted at the current context. • `ALF_ERR_NOMEM`: Out of memory. • `ALF_ERR_FAULT`: Invalid buffer or error handler address (only when it is possible to detect the fault). • `ALF_ERR_GENERIC`: Generic internal errors.

Work block API

The following API definitions are the work block APIs.

Data structures

alf_wb_handle_t

This data structure refers to the work block being constructed by the control node.

alf_wb_create

NAME

alf_wb_create - Creates a new work block for the specified compute task.

SYNOPSIS

int alf_wb_create(alf_task_handle_t task_handle, ALF_WORK_BLOCK_TYPE_T work_block_type, unsigned int repeat_count, alf_wb_handle_t *p_wb_handle);

Parameters

`p_wb_handle` `[OUT]`	The pointer to a buffer where the created handle is returned. The contents are not modified if this call fails.
`task_handle` `[IN]`	The handle to the compute task.
`work_block_type` `[IN]`	The type of work block to be created. Choose from the following types: • `ALF_WB_SINGLE`: Creates a single-use work block • `ALF_WB_MULTI`: Creates a multi-use work block. This work block type is only supported when the task is created with the `ALF_PARTITION_ON_ACCEL` attribute.
`repeat_count` `[IN]`	Specifies the number of iterations for a multi-use work block. This parameter is ignored when a single-use work block is created.

DESCRIPTION

This function creates a new work block for the specified computing task. The work block is added to the work queue of the task and the runtime releases the allocated resources once the work block is processed. The caller can only update the contents of a work block before it is added to the work queue. After the work block is added to the work queue, the lifespan of the data structure is left to the ALF runtime. The ALF runtime is responsible for cleaning up any resource allocated for the work block. This API can only be called before `alf_task_finalize` is invoked. After the `alf_task_finalize` is called, further calls to this API return an error.

RETURN VALUE

0	Success.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_PERM`: Operation not allowed in current context. For example, the task has already been finalized or the work block has been enqueued. • `ALF_ERR_BADF`: Invalid task handle. • `ALF_ERR_NOMEM`: Out of memory. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_wb_enqueue

NAME

alf_wb_enqueue - Adds the work block to the work queue of the specified task handle.

SYNOPSIS

int alf_wb_enqueue(alf_wb_handle_t wb_handle)

Parameters

`wb_handle [IN]` The handle of the work block to be put into the work queue.

DESCRIPTION

This function adds the work block to the work queue of the specified task handle. The caller can only update the contents of a work block before it is added to the work queue. After it is added to the work queue, you cannot access the `wb_handle`.

RETURN VALUE

0	Success.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid task handle or work block handle • `ALF_ERR_PERM`: Operation not allowed in current context • `ALF_ERR_BUSY`: An internal resource is occupied • `ALF_ERR_GENERIC`: Generic internal errors

alf_wb_parm_add

NAME

alf_wb_parm_add - Adds the given parameter to the parameter and context buffer of the work block in the order that this function is called.

SYNOPSIS

int alf_wb_parm_add(alf_wb_handle_t wb_handle, void *pdata, unsigned int size_of_data, ALF_DATA_TYPE_T data_type, unsigned int address_alignment)

Parameters

`wb_handle [IN]`	The work block handle.
`pdata [IN]`	A pointer to the data to be copied.
`size_of_data [IN]`	The size of the data in units of the data type.
`data_type [IN]`	The type of data. This value is required if data endianess conversion is necessary when moving the data.
`address_alignment [IN]`	Power of 2 byte alignment of 2 $^{address\_alignment}$. The valid range is from 0 to 16. A zero indicates a byte-aligned address. An 8 indicates alignment on 256 byte boundaries.

DESCRIPTION

This function adds the given parameter to the **parameter and context buffer** of the work block in the order that this function is called. The starting address is from offset zero. The added data is copied to the internal parameter and context buffer immediately. The relative address of the data can be aligned as specified. For a specific work block, additional calls to this API return an error after the work block is put into the work queue by calling the `alf_wb_enqueue` function.

RETURN VALUE

0	Success.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_PERM`: Operation not allowed in current context. • `ALF_ERR_BADF`: Invalid task handle or work block handle. • `ALF_ERR_NOBUFS`: Some internal resource is occupied. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_wb_dtl_begin

NAME

alf_wb_dtl_begin - Marks the beginning of a data transfer list for the specified target `buffer_type`.

SYNOPSIS

int alf_wb_dtl_begin (alf_wb_handle_t wb_handle, ALF_BUF_TYPE_T buffer_type, unsigned int offset_to_accel_buf);

Parameters

`wb_handle [IN]`	The work block handle.
`buffer_type [IN]`	The type of the buffer. Possible values are: • `ALF_BUF_IN`: Input to the input only buffer • `ALF_BUF_OUT`: Output from the output only buffer • `ALF_BUF_OVL_IN`: Input to the overlapped buffer • `ALF_BUF_OVL_OUT`: Output from the overlapped buffer • `ALF_BUF_OVL_INOUT`: In/out to/from the overlapped buffer
`offset_to_accel_buf [IN]`	Offset of the target buffer on the accelerator.

DESCRIPTION

This function marks the beginning of a data transfer list for the specified target `buffer_type`. Further calls to function `alf_wb_dtl_entry_add` refers to the currently opened data transfer list. You can create multiple data transfer lists per buffer type, however, only one data transfer list is opened for entry at any time for a specific work block there can be no nesting of data transfer list.

RETURN VALUE

0	Success.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_PERM`: Operation not allowed. • `ALF_ERR_BADF`: Invalid work block handle. • `ALF_ERR_2BIG`: The offset to the accelerator buffer is larger than the size of the buffer. • `ALF_ERR_NOSYS`: The specified I/O type feature is not supported. • `ALF_ERR_BADR`: The requested buffer is not defined in the task context. • `ALF_ERR_GENERIC`: Generic internal errors. • `ALF_ERR_NOBUFS`: The internal data buffer is used up.

alf_wb_dtl_entry_add

NAME

alf_wb_dtl_entry_add - Adds an entry to the input or output data transfer lists of a single use work block.

SYNOPSIS

int alf_wb_dtl_entry_add (alf_wb_handle_t wb_handle, void* host_addr, unsigned int size, ALF_DATA_TYPE_T data_type);

Parameters

`wb_handle [IN]`	The work block handle
`host_address [IN]`	The pointer (EA) to the data in remote memory
`size [IN]`	The size of the data in units of the data type
`data_type [IN]`	The type of data, this value is required if data endianess conversion is necessary when doing the data movement

DESCRIPTION

This function adds an entry to the input or output data transfer lists of a single use work block. The entry describes a single piece of data transferred from and to the remote memory. For a specific work block, further calls to this API return errors after the work block is put to work queue by calling `alf_wb_enqueue`.

For a specific work block, further calls to this API return error after the work block is put to work queue by calling `alf_wb_enqueue`. If the work block's task is associated with a dataset, the specified buffer with **host_addr** and **size** must be contained within the dataset. Adding a dtl entry describing a buffer that is outside the associated dataset returns a `ALF_ERR_PERM` error.

This function can only be called if the task descriptor associated with the work block's task is created with the task descriptor attribute `ALF_TASK_DESC_PARTITION_ON_ACCEL` set to false.

RETURN VALUE

0	Success.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_PERM`: Operation not allowed. • `ALF_ERR_BADF`: Invalid work block handle. • `ALF_ERR_2BIG`: Trying to add too many lists. • `ALF_ERR_NOBUFS`: The amount of data to move exceeds the maximum buffer size. • `ALF_ERR_FAULT`: Invalid host address (if it can be detected). • `ALF_ERR_GENERIC`: Generic internal errors.

alf_wb_dtl_end

NAME

alf_wb_dtl_end - This function marks the ending of a data transfer list.

SYNOPSIS

int alf_wb_dtl_end (alf_wb_handle_t wb_handle);

Parameters

`wb_handle [IN]`	The work block handle

DESCRIPTION

This function marks the ending of a data transfer list.

RETURN VALUE

0	Success.
less than 0	Errors occurred: • `ALF_ERR_PERM`: Operation not allowed. • `ALF_ERR_BADF`: Invalid work block handle.

Data set API

The following API definitions are the data set APIs.

alf_dataset_create

NAME

alf_dataset_create - Creates a dataset.

SYNOPSIS

int alf_dataset_create(alf_handle_t alf_handle, alf_dataset_handle_t * p_dataset_handle);

Parameters

`alf_handle[in]`	Handle to the ALF runtime
`p_dataset_handle[out]`	Handle to the dataset

DESCRIPTION

This function creates a dataset.

RETURN VALUE

0	Success
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid ALF handle • `ALF_ERR_GENERIC`: Generic internal errors

alf_dataset_buffer_add

NAME

alf_dataset_buffer_add - Adds a data buffer to the data set.

SYNOPSIS

int alf_dataset_buffer_add(alf_dataset_handle_t dataset, void *buffer, unsigned long long size, ALF_CACHE_ACCESS_MODE_T access_mode);

Parameters

`buffer`	Address of the buffer to be added
`size`	Size of the buffer
`access mode`	Access mode for the buffer. A buffer can have either of the following access modes: • `ALF_DATASET_READ_ONLY`: The dataset is read-only. Work blocks referencing the data in this buffer cannot update this buffer as an output buffer. • `ALF_DATASET_WRITE_ONLY`: The dataset is write-only. Work blocks referencing the data in this buffer as input data result in indeterminate behavior. • `ALF_DATASET_READ_WRITE`: The dataset allows both read and write access. Work blocks can use this buffer as input buffers and output buffers and/or in out buffers.

DESCRIPTION

This function adds a data buffer to the data set.

RETURN VALUE

0	Success
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid ALF handle • `ALF_ERR_PERM`: The API call is not permitted with the current calling context. The dataset has been associated with a task and thus closed from further buffer additions. • `ALF_ERR_GENERIC`: Generic internal errors

alf_dataset_destroy

NAME

alf_dataset_destroy - Destroys a given data set.

SYNOPSIS

int alf_dataset_destroy(alf_dataset_handle_t dataset_handle);

Parameters

`dataset_handle`	Handle to the dataset

DESCRIPTION

This function destroys a given dataset. Further references to the dataset result in indeterminate behaviors. Further references to the data within a dataset are still valid. You cannot destroy a dataset if there are still running tasks associated with a dataset.

RETURN VALUE

0	Success
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_BADF`: Invalid ALF handle. • `ALF_ERR_PERM`: The API call is not permitted with the current calling context. The dataset has been associated with a task and thus closed from further buffer additions. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_task_dataset_associate

NAME

alf_task_dataset_associate - Associates a given task with a dataset.

SYNOPSIS

int alf_task_dataset_associate(alf_task_handle_t task, alf_dataset_handle_t dataset);

Parameters

`dataset_handle`	Handle to dataset
`task_handle`	Handle to the task

DESCRIPTION

This function associates a given task with a dataset. This function can only be called before any work block is enqueued for the task. After a task is associated with a dataset, all subsequent work blocks created and enqueued for this task cannot reference data outside the dataset.

After a task is associated with a dataset, further calls to `alf_data_buffer_add` results in error.

After a task is associated with a dataset, the host application program can only use the data after `alf_task_wait` is called and returned.

RETURN VALUE

0	Success
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid ALF handle • `ALF_ERR_PERM`: The API call is not permitted with the current calling context. The dataset has been associated with a task and thus closed from further buffer additions. • `ALF_ERR_SRCH`: Already destroyed task handle • `ALF_ERR_GENERIC`: Generic internal errors

Chapter 16. Accelerator API

The following API definitions are the accelerator APIs.

Computational kernel function exporting macros

The ALF MPMD programming model supports multiple computational kernels in a single accelerator execution image. To allow the ALF runtime to differentiate between different functions for different kernels, you need to export these functions to the ALF runtime. Some macros are provided to make sure the function exporting can be performed in a platform-neutral way. In each accelerator side execution image, there must be at least one computational kernel API exporting definition section. However the maximum allowed number of sections is platform-dependent.

The following example shows how these macros are used.

```
/* API implementations for task "foo" */
int foo_comp_kernel(...) {...}
int foo_input_prepare(...) {...}
int foo_output_prepare(...) {...}
int foo_ctx_setup(...) {...}
int foo_ctx_merge(...) {...}

/* API implementations for task "bar" */
int bar_comp_kernel(...) {...}
int bar_input_prepare(...) {...}
int bar_output_prepare(...) {...}
int bar_ctx_setup(...) {...}
int bar_ctx_merge(...) {...}

/* API exporting definition section */
ALF_ACCEL_API_LIST_BEGIN

/* for task "foo" */
ALF_ACCEL_EXPORT_API ("foo_comp_kernel", foo_comp_kernel);
ALF_ACCEL_EXPORT_API ("foo_input_prepare", foo_input_prepare);
ALF_ACCEL_EXPORT_API ("foo_output_prepare", foo_output_prepare);
ALF_ACCEL_EXPORT_API ("foo_ctx_setup", foo_ctx_setup);
ALF_ACCEL_EXPORT_API ("foo_ctx_merge", foo_ctx_merge);

/* for tas "bar" */
ALF_ACCEL_EXPORT_API ("bar_comp_kernel", bar_comp_kernel);
ALF_ACCEL_EXPORT_API ("bar_input_prepare", bar_input_prepare);
ALF_ACCEL_EXPORT_API ("bar_output_prepare", bar_output_prepare);
ALF_ACCEL_EXPORT_API ("bar_ctx_setup", bar_ctx_setup);
ALF_ACCEL_EXPORT_API ("bar_ctx_merge", bar_ctx_merge);

ALF_ACCEL_EXPORT_API_LIST_END
```

ALF_ACCEL_EXPORT_API_LIST_BEGIN

This macro declares the beginning of computational kernel API exporting definition section. This macro must be the first statement of the definition section.

ALF_ACCEL_EXPORT_API_LIST_END

This macro declares the ending of computational kernel API exporting definition section. This macro must be the last statement of the definition section.

ALF_ACCEL_EXPORT_API

NAME

ALF_ACCEL_EXPORT_API - Declares one entry of the computing kernel API exporting definition section.

SYNOPSIS

ALF_ACCEL_EXPORT_API(const char *p_api_name, int (*p_api)())

Parameters

`p_api_name[IN]`	The string constant that uniquely identifies the exported API. It is recommended to be just the same as the correspondent function identifier.
`p_api[IN]`	The exported function entry pointer.

DESCRIPTION

This macro declares one entry of the computing kernel API exporting definition section. The ALF runtime locates the entry address of the user-implemented computing kernel functions based on information provided by the corresponding entries.

User-provided computational kernel APIs

This section lists the prototypes of accelerator APIs that you need to implement. Some of these functions are optional functions, which you do not need to implement if not required.

Note: For documentation purposes, names are provided for these different prototype APIs. However, you can choose your own function names for your implementations of these functions.

alf_accel_comp_kernel

NAME

alf_accel_comp_kernel - Computes the work blocks.

SYNOPSIS

int alf_accel_comp_kernel(void* p_task_ctx, void *p_parm_ctx_buffer, void *p_input_buffer, void *p_output_buffer, void* p_inout_buffer, unsigned int current_iter, unsigned int num_iter);

Parameters

`p_task_context [IN]`	A pointer to the local memory block where the task context buffer is kept.
`p_parm_ctx_data [IN]`	A pointer to the local memory block where the parameter and context data are kept.
`p_input_buffer [IN]`	A pointer to the local memory block where the input data is loaded.
`p_output_buffer [IN]`	A pointer to the local memory block where the output data is written.
`p_inout_buffer [IN]`	A pointer to the accelerator memory block where the in/out buffers are located.
`current_iter [IN]`	The current iteration count of multi-use work blocks. This value starts at 0. For single-use work blocks, this value is always 0.
`num_iter [IN]`	The total number of iterations of multi-use work blocks. For single-use work blocks, this value is always 1.

DESCRIPTION

This is the computational kernel that does the computation of the work blocks. The ALF runtime ensures that all input data are available before invoking this call. You must provide an implementation for this function.

RETURN VALUE

0	The computation finished correctly.
less than `0`	An error occurred during the computation. The error code is passed back to you to be handled.

alf_accel_input_dtl_prepare

NAME

alf_accel_input_dtl_prepare - Defines the data transfer lists for input data.

SYNOPSIS

int alf_accel_input_dtl_prepare (void* p_task_context, void *p_parm_context, void *p_dtl, unsigned int current_iter, unsigned int num_iter);

Parameters

`p_task_context[IN]`	Pointer to the task context buffer in accelerator memory.
`p_parm_ctx_buffer[IN]`	Pointer to the work block parameter context buffer in accelerator memory.
`p_dtl[IN]`	Pointer the data transfer list the generated data transfer list should be saved.
`current_iter[IN]`	The current iteration count of multi-use work blocks. This value starts at 0. For single-use work blocks, this value is always 0.
`num_iter[IN]`	The total number of iterations of multi-use work blocks. For single-use work blocks, this value is always 1.

DESCRIPTION

This function is called by the ALF runtime when it needs the accelerator to define the data transfer lists for input data. One important point to consider is that because the ALF framework may do double buffering, the function only refers to the information provided by the `p_parm_ctx_buffer`. This function should generate the data transfer lists for the input buffer (`ALF_BUF_IN`), the overlapped input buffer (`ALF_BUF_OVL_IN`), and the overlapped I/O buffer (`ALF_BUF_OVL_INOUT`) when these buffers are enabled. For the overlapped I/O buffer (`ALF_BUF_OVL_INOUT`), the data transfer list generated in this function is reused by the runtime to push the data back to host memory.

This function is an optional function. It is only called if the task descriptor sets the `ALF_TASK_DESC_PARTITION_ON_ACCEL` to true. When this attribute is not set or set to false, you can choose not to implement this API when the programming environment supports weak link or to implement an empty function that returns zero when weak link is not supported.

RETURN VALUE

0	The computation finished correctly.
less than 0	An error occurred during the call. The error code is passed back to you to be handled.

alf_accel_output_dtl_prepare

NAME

alf_accel_output_dtl_prepare - Defines the partition of output data.

SYNOPSIS

int alf_accel_output_dtl_prepare (void* p_task_context, void *p_parm_ctx_buffer, void *p_io_container, unsigned int current_iter, unsigned int num_iter);

Parameters

`p_task_context[IN]`	Pointer to the task context buffer in accelerator memory.
`p_parm_ctx_buffer[IN]`	Pointer to the work block parameter context in accelerator memory.
`p_dt_list_buffer[IN]`	Pointer to the buffer where the generated data transfer list should be saved.
`current_iter[IN]`	The current iteration count of multi-use work blocks. This value starts at 0. For single-use work blocks, this value is always 0.
`num_iter[IN]`	The total number of iterations of multi-use work blocks. For single-use work blocks, this value is always 1.

DESCRIPTION

This function is called by the ALF runtime when it needs the accelerator to define the partition of output data. Because the ALF may be doing double buffering, the function should only refer to the information provided by the `p_parm_ctx_buffer`. This function generates the data transfer lists for the output buffer (`ALF_BUF_OUT`) and the overlapped output buffer (`ALF_BUF_OVL_OUT`) when these buffers are enabled.

This function is only called if the task descriptor sets the `ALF_TASK_DESC_PARTITION_ON_ACCEL` to true. When this attribute is not set or set false, you can choose not to implement this API when the programming environment supports weak link or to implement an empty function that return zero when weak link is not supported.

RETURN VALUE

0	The computation finished correctly.
less than 0	An error occurred during the call. The error code is passed back to you to be handled.

alf_accel_task_context_setup

NAME

alf_accel_task_context_setup - Initializes a task.

SYNOPSIS

int alf_accel_task_context_setup (void* p_task_context);

Parameters

`p_task_context [IN/OUT]`	Pointer to task context in accelerator memory.

DESCRIPTION

This function is called by the ALF runtime when a task starts running on an accelerator. The runtime loads the initial task context to the local memory and calls this function to do some task instance specific initialization.

The ALF runtime only invokes this API when the task has a task context. When the task does not have a task context or the application does not need extra setup of the initial context, you can choose not to implement this API when the programming environment supports weak link or to implement an empty function that returns zero when weak link is not supported.

RETURN VALUE

0	The API call finished correctly.
less than 0	An error happened during the call. The error code is passed back to you to be handled.

alf_accel_task_context_merge

NAME

alf_accel_task_context_merge - Merges the context after a task has stopped running.

SYNOPSIS

int alf_accel_task_context_merge (void* p_task_context_to_be_merged, void* p_task_context);

Parameters

`p_task_context_to_merge[IN]`	Pointer to the local memory block where the to be merged task context buffer is kept.
`p_task_context[IN/OUT]`	Pointer to the local memory block where the to be target task context buffer is kept.

DESCRIPTION

This function is called by the ALF runtime when a task stops running on an accelerator. The runtime loads the corresponding task context to the memory of an accelerator that is running this task and calls this function to do the context merge.

The ALF runtime only invokes this API only when the task has a task context. If the task does not have a task context or the application does not need to do context merge, you can choose not to implement this API when the programming environment supports weak link or to implement an empty function that returns zero when weak link is not supported.

RETURN VALUE

0	The API call finishes correctly.
less than 0	An error occurred during the call. The error code is passed back to you to be handled.

Runtime APIs

This section lists the APIs that accelerator side ALF runtime provides.

alf_accel_num_instances

NAME

alf_accel_num_instances - Returns the number of instances that are running this computational kernel.

SYNOPSIS

int alf_accel_num_instances (void);

Parameters
None

DESCRIPTION

This function returns the number of instances that are currently executing this computational kernel. This function should only be used when a task is created with the task attribute ALF_TASK_ATTR_SCHED_FIXED. If user calls this function without ALF_TASK_ATTR_SCHED_FIXED, the number returned might change from one invocation to the next as the ALF runtime dynamically loads and unloads task instances.

RETURN VALUE

>0	number of accelerators that are executing this compute task
less than 0	Internal error

alf_instance_id

NAME

alf_instance_id - Returns the number of instances that are running this computational kernel.

SYNOPSIS

int alf_instance_id (void);

Parameters
None

DESCRIPTION

This function returns the current instance ID of the task. This ID ranges from 0 to `alf_accel_num_instances`.

RETURN VALUE

>=0	Returns the ID of the current accelerator. This is guaranteed to be unique within the reserved accelerators for ALF runtime
less than 0	Internal error

ALF_ACCEL_DTL_BEGIN

NAME

ALF_ACCEL_DTL_BEGIN - Marks the beginning of a data transfer list for the specified target `buffer_type`.

SYNOPSIS

ALF_ACCEL_DTL_BEGIN (void* p_dtl, ALF_IO_BUF_TYPE_T buf_type, unsigned int offset);

Parameters

`p_dtl[IN/OUT]`	Pointer to buffer for the data transfer list data structure.

`buf_type`	`ALF_BUF_IN`
	`ALF_BUF_OUT`
	`ALF_OVL_IN`
	`ALF_OVL_OUT`
	`ALF_OVL_INOUT`

`offset[IN]`	Offset to the input or output buffer pointer in local memory to which the data transfer list refers to.

DESCRIPTION

This utility marks the beginning of a data transfer list for the specified target `buffer_type`. Further calls to function `ALF_ACCEL_DTL_ENTRY_ADD` refer to the currently opened data transfer list. You can create multiple data transfer lists per buffer type. However, only one data transfer list is opened for entry at any time.

Note: This API is for accelerator node side to generate the data transfer list entries. It may be implemented as macros on some platforms.

RETURN VALUE

None.

ALF_ACCEL_DTL_ENTRY_ADD

NAME

ALF_ACCEL_DTL_ENTRY_ADD - Fills the data transfer list entry.

SYNOPSIS

ALF_ACCEL_DTL_ENTRY_ADD (void *p_dtl, unsigned int data_size, ALF_DATA_TYPE_T data_type, alf_data_addr64_t p_host_address);

Parameters

`p_dtl[IN]`	Pointer to buffer for the data transfer list data structure.
`data_size[IN]`	Size of the data in unit of the data type.
`data_type[IN]`	The type of data. This value is required if data endianess conversion is necessary when moving the data.
`host_address[IN]`	Address of the host memory.

DESCRIPTION

This function fills the data transfer list entry.

This API is for the accelerator node side to generate the data transfer list entries. It can be implemented as macros on some platforms.

Note: This API is for accelerator node side to generate the data transfer list entries. It can be implemented as macros on some platforms.

RETURN VALUE

None.

ALF_ACCEL_DTL_END

NAME

ALF_ACCEL_DTL_END - Marks the ending of a data transfer list.

SYNOPSIS

ALF_ACCEL_DTL_END(void* p_dtl);

Parameters

p_dtl[IN]	Pointer to buffer for the data transfer list data structure.

DESCRIPTION

This utility marks the ending of a data transfer list.

RETURN VALUE

None.

Part 5. Appendixes

Appendix A. Examples

The following examples are described in this section:

Basic examples

This section describes the following basic examples:

- "Matrix add - host data partitioning example." This example includes the source code.
- "Matrix add - accelerator data partitioning example" on page 112.

Matrix add - host data partitioning example

In this example, two large matrices are added together using ALF. The problem can be expressed simply as:

```
A[m,n] + B[m,n] = C[m,n]
```

where `m` and `n` are the dimensions of the matrices.

This simple example demonstrates how to:

- Use task descriptor
- Start a task on the accelerators
- Create and add a work block to a task
- Exit the ALF runtime environment correctly

You can also use this sample as a template to build a more complicated application.

In this example, the host application:

- Initializes the ALF runtime environment
- Creates a task descriptor
- Creates a task based on that task descriptor
- Creates work blocks with the appropriate data transfer lists which start invocations of the computational kernel on the accelerator
- Waits for the computational kernel to finish and exits

The accelerator application includes a simple computational kernel that computes the addition of the two matrices.

The scalar code to add two matrices for a uni-processor machine is provided below:

```
float mat_a[NUM_ROW][NUM_COL];
float mat_b[NUM_ROW][NUM_COL];
float mat_c[NUM_ROW][NUM_COL];
int main(void)
{
  int i,j;
  for (i=0; i<NUM_ROW; i++)
     for (j=0; j<NUM_COL; j++)
        mat_c[i][j] = mat_a[i][j] + mat_b[i][j];
  return 0;
}
```

An ALF host program can be logically divided into several sections:

- Initialization
- Task setup
- Work block set up
- Task wait and exit

Source code

The following code listings only show the relevant sections of the code. For a complete listing, refer to the ALF samples directory

`matrix_add/STEP1a_partition_scheme_A`

Initialization

The following code segment shows how ALF is initialized and accelerators allocated for a specific ALF runtime.

```
alf_handle_t alf_handle;
unsigned int nodes;

/* initializes the runtime environment for ALF*/
alf_init(&config_parms;, &alf_handle;);

/* get the number of SPE accelerators available for from the Opteron */
rc = alf_query_system_info(alf_handle, ALF_QUERY_NUM_ACCEL, ALF_ACCEL_TYPE_SPE, &nodes;);

/* set the total number of accelerator instances (in this case, SPE) */
/* the ALF runtime will have during its lifetime */
rc = alf_num_instances_set (alf_handle, nodes);
```

Task setup

The next section of an ALF host program contains information about the description of a task and the creation of the task runtime. The `alf_task_desc_create` function creates a task descriptor. This descriptor can be used multiple times to create different executable tasks. The function `alf_task_create` creates a task to run an SPE program with the name *spe_add_program*.

```
/* variable declarations */
alf_task_desc_handle_t task_desc_handle;
alf_task_handle_t task_handle;
const char* spe_image_name;
const char* library_path_name;
const char* comp_kernel_name;

/* describing a task that's executable on the SPE*/
```

```
alf_task_desc_create(alf_handle, ALF_ACCEL_TYPE_SPE, &task_desc_handle;);
alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_TSK_CTX_SIZE, 0);
alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_WB_PARM_CTX_BUF_SIZE,
sizeof(add_parms_t));
alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_WB_IN_BUF_SIZE, H * V * 2
sizeof(float));
alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_WB_OUT_BUF_SIZE, H * V *
sizeof(float));
alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_NUM_DTL_ENTRIES, 8);
alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_MAX_STACK_SIZE, 4096);
/* providing the SPE executable name */
alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_IMAGE_REF_L,
(unsigned long long) spe_image_name);
alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_LIBRARY_REF_L,
(unsigned long) library_path_name);
alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_KERNEL_REF_L,
(unsigned long) comp_kernel_name);
```

Work block setup

This section shows how work blocks are created. After the program has created the work block, it describes the input and output associated with each work block. Each work block contains the input description for blocks in the input matrices of size `H * V` starting at location `matrix[row][0]` with `H` and `V` representing the horizontal and vertical dimensions of the block.

In this example, assume that the accelerator memory can contain the two input buffers of size `H * V` elements and the output buffer of size `H * V`. The program calls `alf_wb_enqueue()` to add the work block to the queue to be processed. ALF employs an immediate runtime mode. As soon as the first work block is added to the queue, the task starts processing the work block. The function `alf_task_finalize` closes the work block queue.

```
alf_wb_handle_t wb_handle;
add_parms_t parm __attribute__((aligned(128)));
parm.h = H; /* horizontal size of the block */
parm.v = V; /* vertical size of the block */

/* creating work blocks and adding param & io buffer */
for (i = 0; i < NUM_ROW; i += H) {
  alf_wb_create(task_handle, ALF_WB_SINGLE, 0,
&wb_handle);
/* begins a new Data Transfer List for INPUT */
  alf_wb_dtl_set_begin(wb_handle,
ALF_BUF_IN, 0);
/* Add H*V element of mat_a as Input */
  alf_wb_dtl_set_entry_add(wb_handle, &matrix_a[i][0], H * V, ALF_DATA_FLOAT);
 /* Add H*V element of mat_b as Input */
  alf_wb_dtl_set_entry_add(wb_handle, &matrix_b[i][0], H * V, ALF_DATA_FLOAT);
  alf_wb_dtl_set_end(wb_handle);
 /* begins a new Data Transfer List OUTPUT */
  alf_wb_dtl_set_begin(wb_handle, ALF_BUF_OUT, 0);
 /* Add H*V element of mat_c as Output */
  alf_wb_dtl_set_entry_add(wb_handle, &matrix_c[i][0], H * V, ALF_DATA_FLOAT);
  alf_wb_dtl_set_end(wb_handle);
 /* pass parameters H and V to spu */
  alf_wb_parm_add(wb_handle, (void *) (&parm), sizeof(parm), ALF_DATA_BYTE, 0);
 /* enqueuing work block */
  alf_wb_enqueue(wb_handle);
}
alf_task_finalize(task_handle);
```

Task wait and exit

After all the work blocks are on the processing queue, the program waits for the accelerator to finish processing the work blocks. Then `alf_exit()` is called to cleanly exit the ALF runtime environment.

```
/* waiting for all work blocks to be done*/
alf_task_wait(task_handle, -1);
/* exit ALF runtime */
alf_exit(alf_handle, ALF_EXIT_WAIT, -1);
```

Accelerator side

On the accelerator side, you need to provide the actual computational kernel that computes the addition of the two blocks of matrices. The ALF runtime on the accelerator is responsible for getting the input buffer to the accelerator memory before it runs the user-provided `alf_accel_comp_kernel` function. After `alf_accel_comp_kernel` returns, the ALF runtime is responsible for getting the output data back to host memory space. Double buffering or triple buffering is employed as appropriate to ensure that the latency for the input buffer to get into accelerator memory and the output buffer to get to host memory space is well covered with computation.

```
int alf_accel_comp_kernel(void *p_task_context,
void *p_parm_context,
void *p_input_buffer,
         void *p_output_buffer,
void *p_inout_buffer,
         unsigned int current_count,
unsigned int total_count
{
unsigned int i, cnt;
  vector float *sa, *sb, *sc;
  add_parms_t *p_parm = (add_parms_t *)
p_parm_context;
  cnt = p_parm->h * p_parm->v / 4;
  sa = (vector float *) p_input_buffer;
  sb = sa + cnt;
  sc = (vector float *) p_output_buffer;
  for (i = 0; i < cnt; i += 4) {
    sc[i] = spu_add(sa[i], sb[i]);
    sc[i + 1] = spu_add(sa[i + 1], sb[i + 1]);
    sc[i + 2] = spu_add(sa[i + 2], sb[i + 2]);
    sc[i + 3] = spu_add(sa[i + 3], sb[i + 3]);
   }
   return 0;
}
```

Matrix add - accelerator data partitioning example

In this example, the same problem as presented in “Matrix add - host data partitioning example” on page 109 is solved, adding two large matrices using ALF. The code remains the same on the host except for the work block creation. The code also needs to specify that it uses accelerator data partitioning in the task descriptor.

An implementation for the `alf_accel_input_dtl_prepare` and `alf_accel_output_dtl_prepare` functions is also required.

Task context examples

This section describes the following task context examples:

- "Table lookup example"
- "Min-max finder example" on page 115
- "Multiple vector dot products" on page 116

Table lookup example

This example shows how the task context buffer is used as a large lookup table to convert the 16 bit input data to 8 bit output data.

The lookup table has 65536 entries defined:

```
For all -32768 <= in <32768
              /  0,        in < -4096
Table(in) = |   in/32     -4096 <=in<4096
              \  255,      in >= 4096
```

The following is the stripped down code list. The routines of less interest have been removed to allow you to focus on the key features. Because the task context buffer (the lookup table) is already initialized by the host code and the table is used as read-only data, you do not need the context setup and context merge functions on the accelerator side.

Data structures shared by the host and accelerator

The following code segment shows the data structures shared by both the host and the accelerators. The `my_task_context_t` data structure contains the lookup table. The `my_wb_parms_t` data structure represents the parameter and context data for each work block.

```
/* ---------------------------------------------- */
/* data structures shared by host and accelerator */
/* ---------------------------------------------- */
typedef struct _my_task_context_t
{
    alf_data_byte_t table[65536];
} my_task_context_t;

typedef struct _my_wb_parms_t
{
    alf_data_uint32_t num_data;  /* number of data in this WB */
} my_wb_parms_t;
```

Task descriptor setup

The following code segment shows how the task descriptor is set up for this application. The task context-related information marked in **bold** in the code.

```
alf_task_desc_create(alf_handle, 0, &task_desc_handle;);
  /* set up the task descriptor ... ... */
  /* the computing kernel name */
  alf_task_desc_set_int64(task_desc_handle,
     ALF_TASK_DESC_ACCEL_KERNEL_REF_L, "comp_kernel");
  /* the task context buffer size */
  alf_task_desc_set_int32(task_desc_handle,
     ALF_TASK_DESC_TSK_CTX_SIZE, sizeof(my_task_context_t));
  /* the work block parm buffer size */
  alf_task_desc_set_int32(task_desc_handle,
```

```
    ALF_TASK_DESC_WB_PARM_CTX_BUF_SIZE, sizeof(my_wb_parms_t));
  /* the input buffer size */
  alf_task_desc_set_int32(task_desc_handle,
     ALF_TASK_DESC_WB_IN_BUF_SIZE,
     PART_SIZE*sizeof(alf_data_int16_t));
  /* the output buffer size */
  alf_task_desc_set_int32(task_desc_handle,
     ALF_TASK_DESC_WB_OUT_BUF_SIZE,
     PART_SIZE*sizeof(alf_data_byte_t));
  /* the task context entry */
  alf_task_desc_ctx_entry_add(task_desc_handle, ALF_DATA_BYTE,
    sizeof(my_task_context_t)/sizeof(alf_data_byte_t));
```

Work block setup

The following code segment shows the code for work block creation.

```
  /* creating wb and adding param & io buffer */
  for (i = 0; i < NUM_DATA; i += PART_SIZE)
  {
    alf_wb_create(task_handle, ALF_WB_SINGLE, 0, &wb_handle);
    alf_wb_dtl_begin(wb_handle, ALF_BUF_IN, 0); /* input */
     alf_wb_dtl_entry_add(wb_handle, pcm16_in+i,
            PART_SIZE, ALF_DATA_INT16);
    alf_wb_dtl_end(wb_handle);
    alf_wb_dtl_begin(wb_handle, ALF_BUF_OUT, 0); /* output */
     alf_wb_dtl_entry_add(wb_handle, pcm8_out+i,
            PART_SIZE, ALF_DATA_BYTE);
    alf_wb_dtl_end(wb_handle);
    wb_parm.num_data = PART_SIZE;
     alf_wb_parm_add(wb_handle, (void *)&wb_parm, /* wb parm */
        sizeof(wb_parm)/sizeof(unsigned int), ALF_DATA_INT32, 0);
    alf_wb_enqueue(wb_handle);
```

Accelerator code

The following code is the accelerator side code. The section of the code that modifies the task context is marked in **bold**.

```
/* ------------------------------------------- */
/* the accelerator side code                   */
/* ------------------------------------------- */
/* the computation kernel function */
int comp_kernel(void *p_task_context, void *p_parm_ctx_buffer,
                   void *p_input_buffer, void *p_output_buffer,
                   void *p_inout_buffer, unsigned int current_count,
                   unsigned int total_count)
{
    my_task_context_t *p_ctx = (my_task_context_t *) p_task_context;
    my_wb_parms_t *p_parm = (my_wb_parms_t *) p_parm_ctx_buffer;

    alf_data_int16_t *in = (alf_data_int16_t *)p_input_buffer;
    alf_data_byte_t *out = (alf_data_byte_t *)p_output_buffer;
    unsigned int size = p_parm->num_data;
    unsigned int i;

    // it is just a simple table lookup
    for(i=0;i<size;i++)
    {
       out[i] = p_ctx->table[(unsigned short)in[i]];
    }     return 0;
}
```

Min-max finder example

This example shows how you can use the task context to keep the partial computing results for each task instance and then combine these partial results into the final result.

The example finds the minimum and maximum values in a large data set. The sequential code is a very simple textbook style implementation, it is a linear search across the whole data set, which compares and updates the best known values with each step.

You can use ALF framework to convert the sequential code into a parallel algorithm. The data set must be partitioned into smaller work blocks. These work blocks are then assigned to the different task instances running on the accelerators. Each invocation of a computational kernel on a task instance is to find the maximum or minimum value in the work block assigned to it. After all the work blocks are processed, you have multiple intermediate best values in the context of each task instance. The ALF runtime then calls the context merge function on accelerators to reduce the intermediate results into the final results.

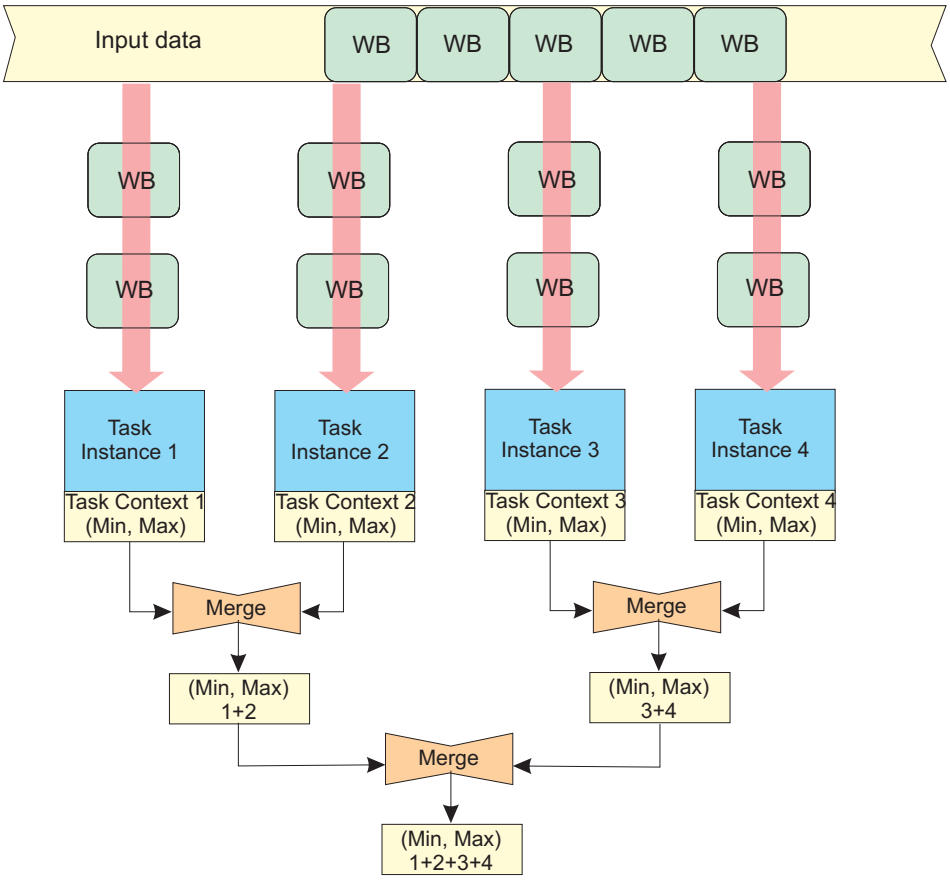

Figure 13. Min-max finder example

Source code

You can find the source code in the sample directory `task_context/min_max`.

Computational kernel

The following code section shows the computational kernel for this application. The computational kernel finds the maximum and minimum values in the provided input buffer then updates the `task_context` with those values.

```
/* ------------------------------------------- */
/* the accelerator side code                   */
/* ------------------------------------------- */
/* the computation kernel function */
int comp_kernel(void *p_task_context, void *p_parm_ctx_buffer,
                   void *p_input_buffer, void *p_output_buffer,
                   void *p_inout_buffer, unsigned int current_count,
                   unsigned int total_count)
{
    my_task_context_t *p_ctx = (my_task_context_t *) p_task_context;
    my_wb_parms_t *p_parm = (my_wb_parms_t *) p_parm_ctx_buffer;

    alf_data_int32_t *a = (alf_data_int32_t *)p_input_buffer;
    unsigned int size = p_parm->num_data;
    unsigned int i;

    /* update the best known values in context buffer */
    for(i=0;i<size;i++) {
        if(a[i]>p_ctx->max)
            p_ctx->max = a[i];
        else if(a[i]<p_ctx->min)
            p_ctx->min = a[i];
    }     return 0;
}
```

Task context merge

The following code segment shows the `context_merge` function for this application. This function is automatically invoked by the ALF runtime after all the task instances have finished processing all the work blocks. The final minimum and maximum values stored in the task context per task instance are merged through this function.

```
/* the context merge function */
int ctx_merge(void* p_task_context_to_be_merged,
void* p_task_context)
{
    my_task_context_t *p_ctx = (my_task_context_t *) p_task_context;
my_task_context_t *p_mgr_ctx = (my_task_context_t *)
 p_task_context_to_be_merged;

    if(p_mgr_ctx->max > p_ctx->max)
        p_ctx->max = p_mgr_ctx->max;
    if(p_mgr_ctx->min < p_ctx->min)
                p_ctx->min = p_mgr_ctx->min;    return 0;
}
```

Multiple vector dot products

This example shows how to use the bundled work block distribution together with the task context to handle situations where the work block can not hold the partitioned data because of a local memory size limit. The example calculates the dot product of two lists of large vectors as:

Given two lists of vectors $\mathbf{A} = \{A_1, A_2, A_3, \ldots, A_m\}$ and $\mathbf{B} = \{B_1, B_2, B_3, \ldots, B_m\}$, where A_i and B_i are dimension N vectors;
Solve $\mathbf{C} = \{c_1, c_2, c_3, \ldots, c_m\}$, where $c_i = A_i \bullet B_i$.
The dot product "●" operation of two dimension N vectors A and B is defined as $A \bullet B = \sum_{i=1}^{N} a_i \cdot b_i$ where a_i and b_i are members of vector A and B.

The dot product requires the element multiplication values of the vectors to be accumulated. In the case where a single work block can hold the all the data for vector Ai and Bi, the calculation is straight forward.

However, when the size of the vector is too big to fit into a single work block, the straight forward approach does not work. For example, with the Cell BE processor, there are only 256 KB of local memory on the SPE. It is impossible to store two double precision vectors when the dimension exceeds 16384. In addition, if you consider the extra memory needed by double buffering, code storage, and so on, you are only be able to handle two vectors of 7500 double precision float point elements each (7500*8[size of double]*2[two vectors] * 2[double buffer] ≈ 240 KB of local storage). In this case, large vectors must be partitioned to multiple work blocks and each work block can only return the partial result of a complete dot product.

You can choose to accumulate the partial results of these work blocks on the host to get the final result. But this is not an elegant solution and the performance is also affected. The better solution is to do these accumulations on the accelerators and do them in parallel.

ALF provides the following two implementations for this problem:

- "Implementation 1: Making use of task context and bundled work block distribution"
- "Implementation 2: Making use of multi-use work blocks together with task context or work block parameter/context buffers" on page 118, with the limitation that accelerator side data partitioning is required

Source code

The source code for the two implementations is provided for you to compare with the shipped samples in the following directories:

- `task_context/dot_prod` directory: Implementation 1. task context and bundled work block distribution
- `task_context/dot_prod_multi` directory: Implementation 2. multi-use work blocks together with task context or work block parameter/context buffers

Implementation 1: Making use of task context and bundled work block distribution

For this implementation, all the work blocks of a single vector are put into a bundle. All the work blocks in a single bundle are assigned to one task instance in the order of enqueuing. This means it is possible to use the task context to accumulate the intermediate results and write out the final result when the last work block is processed.

The accumulator in task context is initialized to zero each time a new work block bundle starts.

When the last work block in the bundle is processed, the accumulated value in the task context is copied to the output buffer and then written back to the result area.

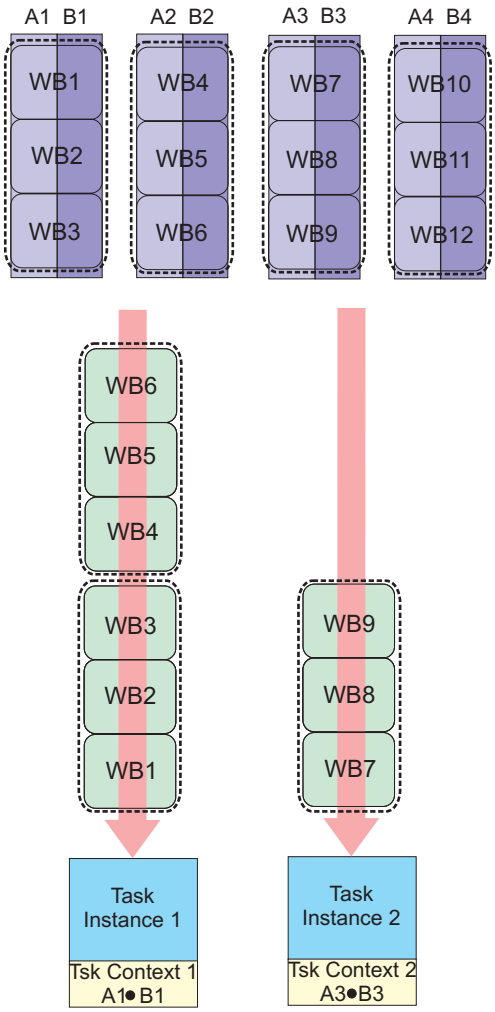

Figure 14. Making use of task context and bundled work block distribution

Implementation 2: Making use of multi-use work blocks together with task context or work block parameter/context buffers

The second implementation is based on multi-use work blocks and work block parameter and context buffers. A multi-use work block is similar to an iteration operation. The accelerator side runtime repeatedly processes the work block until it reaches the provided number of iteration. By using accelerator side data partitioning, it is possible to access different input data during each iteration of the work block. This means the application can be used to handle larger data which a single work block cannot cover due to local storage limitations. Also, the parameter and context buffer of the multi-use work block is kept through the iterations, so you can also choose to keep the accumulator in this buffer, instead of using the task context buffer.

Both methods, using the task context and using multi-use work block are equally valid.

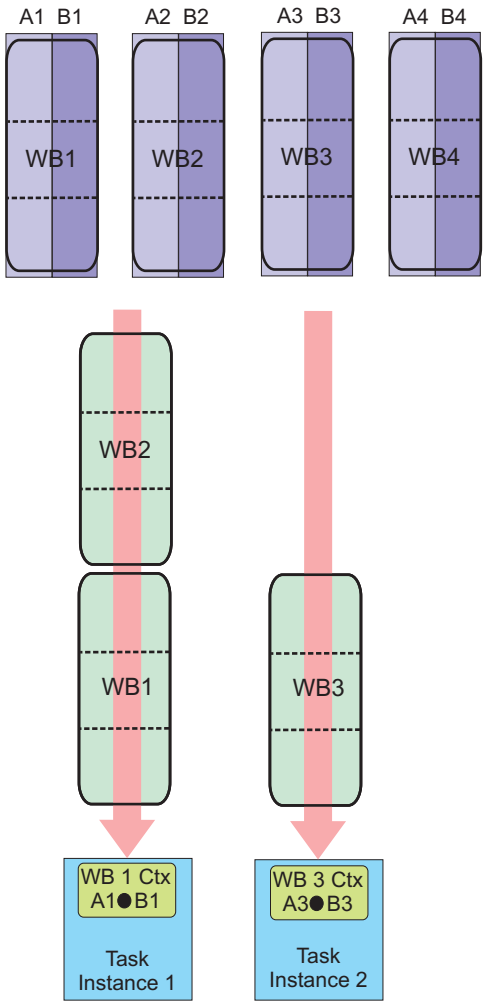

Figure 15. Making use of multi-use work blocks together with task context or work block parameter/context buffers

Overlapped I/O buffer example

The following two simple examples show the usage of overlapped I/O buffers. Both examples do matrix addition.

- The first example implements `C=A+B`, where `A`, `B`, and `C` are different matrices. There are three separate matrices on the host for matrix a, b, and c.
- The second example implements `A=A+B`, where matrix `A` is overwritten by the result. Storage is reserved on the host for matrix a and matrix b. The result of a+b is stored in matrix b.

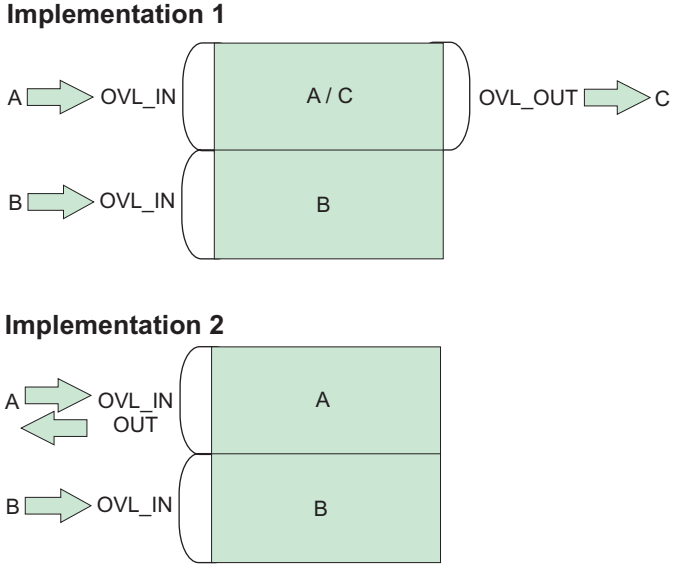

Figure 16. The two overlapped I/O buffer samples

Matrix setup

Note: The code is similar to the matrix_add example, see "Matrix add - host data partitioning example" on page 109. Here only the relevant code listing is shown.

```
/* ------------------------------------------- */
/* matrix declaration for the two cases        */
/* ------------------------------------------- */
#ifdef C_A_B // C = A + B
alf_data_int32_t mat_a[ROW_SIZE][COL_SIZE];  // the matrix a
alf_data_int32_t mat_b[ROW_SIZE][COL_SIZE];  // the matrix b
alf_data_int32_t mat_c[ROW_SIZE][COL_SIZE];  // the matrix c
#else  // A = A + B
alf_data_int32_t mat_a[ROW_SIZE][COL_SIZE];  // the matrix a
alf_data_int32_t mat_b[ROW_SIZE][COL_SIZE];  // the matrix b
#endif
```

Work block setup

This code segment shows the work block creation process for the two cases.

```
  for (i = 0; i < ROW_SIZE; i+=PART_SIZE){
      if(i+PART_SIZE <= ROW_SIZE)
          wb_parm.num_data = PART_SIZE;
      else
          wb_parm.num_data = ROW_SIZE - i;

      alf_wb_create(task_handle, ALF_WB_SINGLE, 0, &wb_handle);

#ifdef C_A_B // C = A + B
      // the input data A and B
      alf_wb_dtl_begin(wb_handle, ALF_BUF_OVL_IN, 0);  // offset at 0
      alf_wb_dtl_entry_add(wb_handle, &mat_a[i][0],
 wb_parm.num_data*COL_SIZE, ALF_DATA_INT32); // A
      alf_wb_dtl_entry_add(wb_handle, &mat_b[i][0],
 wb_parm.num_data*COL_SIZE, ALF_DATA_INT32); // B
      alf_wb_dtl_end(wb_handle);

      // the output data C is overlapped with input data A
      // offset at 0, this is overlapped with A
      alf_wb_dtl_begin(wb_handle, ALF_BUF_OVL_OUT, 0);
      alf_wb_dtl_entry_add(wb_handle, &mat_c[i][0],
```

```
 wb_parm.num_data*COL_SIZE, ALF_DATA_INT32); // C
      alf_wb_dtl_end(wb_handle);
#else  // A = A + B
      // the input and output data A
      alf_wb_dtl_begin(wb_handle, ALF_BUF_OVL_INOUT, 0); // offset 0
      alf_wb_dtl_entry_add(wb_handle, &mat_a[i][0],
 wb_parm.num_data*COL_SIZE, ALF_DATA_INT32); // A
      alf_wb_dtl_end(wb_handle);

      // the input data B is placed after A
// placed after A
      alf_wb_dtl_begin(wb_handle, ALF_BUF_OVL_IN,
 wb_parm.num_data*COL_SIZE*sizeof(alf_data_int32_t));
      alf_wb_dtl_entry_add(wb_handle, &mat_b[i][0],
 wb_parm.num_data*COL_SIZE, ALF_DATA_INT32); // B
      alf_wb_dtl_end(wb_handle);
#endif
      alf_wb_parm_add(wb_handle, (void *)&wb_parm,
 sizeof(wb_parm)/sizeof(unsigned int), ALF_DATA_INT32, 0);
      alf_wb_enqueue(wb_handle);
 }
```

Accelerator code

The accelerator code is shown here. In both cases, the output `sc` can be set to the same location in accelerator memory as `sa` and `sb`.

```
/* ------------------------------------------- */
/* the accelerator side code                   */
/* ------------------------------------------- */
/* the computation kernel function */
int comp_kernel(void *p_task_context, void *p_parm_ctx_buffer,
                    void *p_input_buffer, void *p_output_buffer,
                    void *p_inout_buffer, unsigned int current_count,
                    unsigned int total_count)
{
  unsigned int i, cnt;
  int *sa, *sb, *sc;
  my_wb_parms_t *p_parm = (my_wb_parms_t *) p_parm_context;

  cnt = p_parm->num_data * COL_SIZE;

  sa = (int *) p_inout_buffer;
  sb = sa + cnt;
  sc = sa;

  for (i = 0; i < cnt; i ++)
sc[i] = sa[i] + sb[i];

  return 0;
}
```

Task dependency example

This example shows how task dependency is used in a two stage pipeline application. The problem is a simple simulation.

An object P is placed in the middle of a flat surface with a bounding rectangular box. On each simulation step, the object moves in a random distance in a random direction. It moves back to the initial position when it hits the side walls of the bounding box. The problem is to calculate the number of hits to the four walls in a given time period.

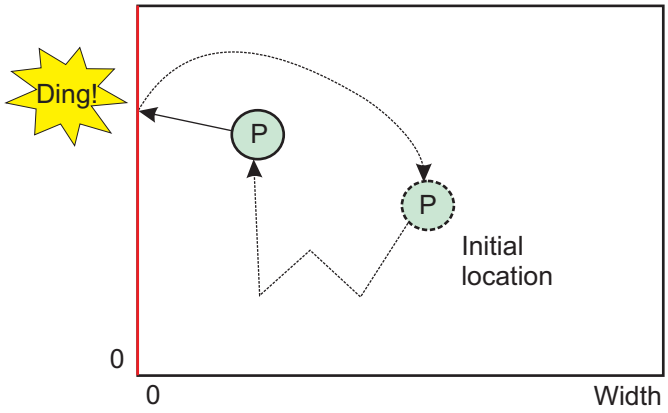

Figure 17. Object P randomly hits the side wall of the bounding box

A two stage pipeline is used to solve the problem so that the random number generation and the simulation can be paralleled:

- The first stage generates random numbers using a pseudo random number generator
- The second stage simulates the movements

Because ALF currently does not support pipeline directly, a pipeline structure is simulated using task dependency. There are two tasks which correspond to the two pipeline stages.

For this problem, each simulation step only needs a small amount of data just as a motion vector. Although ALF does not have a strict limit on how small the data can be, it is better to use larger data blocks for performance considerations. Therefore, the data for thousands of simulation steps is grouped into a single work block.

Stage 1 task: For the stage 1 task, a Lagged Fibonacci pseudo random number generator (PRNG) is used for simplicity. In this example, the algorithm is as follows:

```
Sn=(Sn-j^Sn-k)%232
```

where k > n > 0 and k = 71, j = 65

The algorithm requires a length k history buffer to save the older values. In this implementation, the task context is used for the history buffer. Because no input data is needed, the work block for this task only has output data.

Stage 2 task: For the stage 2 task, the task context is used to save the current status of the simulation including the position of the object and the number of hits to the walls. The work block in this stage only has input data, which are the PRNG results from stage 1.

Another target of pipelining is to overlap the execution of different stages for performance improvement. However, this requires work block level task synchronization between stages, and this is not yet supported by ALF. The alternative approach is to use multiple tasks whereby each task only handles a percentage of the work blocks for the whole simulation.

So there are now two stage tasks. For each chunk of work blocks, the following two tasks are created:

- The stage 1 task generates the random numbers and writes out the results to a temporary buffer
- The stage 2 task reads the random numbers from the temporary buffer to do the simulation

A task dependency is set between the two tasks to make sure the stage 2 task can get the correct results from stage 1 task. Because both the PRNG and the simulation have internal states, you have to pass the states data between the succeeding tasks of the same stage to preserve the states. The approach described here lets the tasks for the same stage share the same task context buffer. Dependencies are used to make sure the tasks access the shared task context in the correct order.

Figure 18 (a) shows the task dependency as described in previous discussions. To further reduce the use of temporary intermediate buffers, you can use double or multi-buffering technology for the intermediate buffers. The task dependency graph for double buffering the intermediate buffers is shown in Figure 18 (b), where a new dependency is added between the n-2th stage 2 task and the nth stage 1 task to make sure the stage 1 task does not overwrite the data that may still be in use by the previous stage 2 task. This is what is implemented in the sample code.

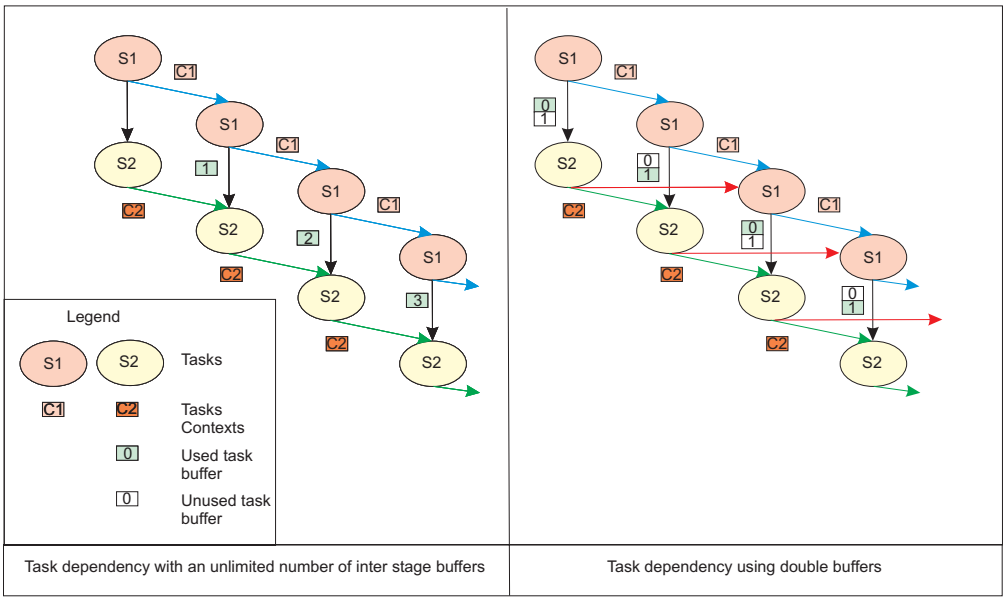

Figure 18. Task dependency examples

Source code

The complete source code can be found in the sample directory `pipe_line`.

Data set example

This example shows the addition of two matrixes and the creation and use of a data set within it.

The data set is created, and is referred to by `dataset_handle`. The data set consists of two buffers:

- mat_a with a size of NUM_ROW*NUM_COL*sizeof(float), and an access_mode of ALF_DATASET_READ_ONLY
- mat_b with a size of NUM_ROW*NUM_COL*sizeof(float), and an access_mode of ALF_DATASET_READ_WRITE

The data set is associated with the task referred to by task_handle.

The following section of code marked in **bold** shows the lines of code that have been added for data set support.

Source code

The source code is provided with the shipped samples in the matrix_add/common/step4_dataset directory.

Sample code: matrix_add.c

```
/* ------------------------------------------------------------ */
/* (C) Copyright 2001,2006,                                     */
/* International Business Machines Corporation,                 */
/*                                                              */
/* All Rights Reserved.                                         */
/* ------------------------------------------------------------ */
/* PROLOG END TAG zYx                                           */

#include <stdio.h>
#include <stdlib.h>
#include <alf.h>
#include "matrix_add.h"
#include "byteswap.h"

#define NUM_ROW 1024
#define NUM_COL 512

#define  H  4
#define  V  NUM_COL

#define PATH_BUF_SIZE 1024  // Max length of path to PPU image
char spu_image_path[PATH_BUF_SIZE];   // Used to hold the complete path to SPU image
char library_name[PATH_BUF_SIZE];     // Used to hold the name of spu library
const char *spu_image_name = "alf_matrix_add_spu";
const char *kernel_name = "comp_kernel";
const char *input_dtl_name = "input_list_prepare";
const char *output_dtl_name = "output_list_prepare";
const char *ctx_setup_name = "context_setup";
const char *ctx_merge_name = "context_merge";

#ifdef _ALF_PLATFORM_HYBRID_
char ppu_image_path[PATH_BUF_SIZE];    // Used to hold the complete path to PPU image
#endif

float mat_a[NUM_ROW][NUM_COL] __attribute__ ((__aligned__(128)));
float mat_b[NUM_ROW][NUM_COL] __attribute__ ((__aligned__(128)));

int main(int argc, char **argv)
{
  alf_handle_t alf_handle;
  alf_task_desc_handle_t task_desc_handle;
  alf_task_handle_t task_handle;
  alf_wb_handle_t wb_handle;
  add_parms_t parm __attribute__ ((aligned(128)));
  int i, j, rc;
  unsigned int nodes;
  int is_spu_dma = 0;
  void *config_parms = NULL;
  alf_dataset_handle_t dataset_handle;

  if (argc == 2) {
    is_spu_dma = atoi(argv[1]);
    if (is_spu_dma < 0)
      is_spu_dma = 0;
  }
```

```
  printf("is_spu_dma: %d\n", is_spu_dma);

  printf("Matrix Addition initializing input.\n");

  /* initialize matrix a b c */
  for (i = 0; i < NUM_ROW; i++) {
    for (j = 0; j < NUM_COL; j++) {
      mat_a[i][j] = (float) (i * NUM_COL + j);
      mat_b[i][j] = (float) ((i * NUM_COL + j) * 2);
    }
  }

  printf("Matrix Addition initialized. Computing results...\n");

  /* begin alf routine */
#ifdef _ALF_PLATFORM_HYBRID_
  alf_sys_config_t_Hybrid hybrid_config_parms;
  hybrid_config_parms.library_path = getenv("HOME");
  hybrid_config_parms.ppe_image_path = getenv("HOME");
  hybrid_config_parms.num_of_ppes = 1;

  config_parms = &hybrid_config_parms;
  sprintf(library_name, "alf_matrix_add_step4_hybrid_spu64.so");
#else
  alf_sys_config_t_CBEA cell_config_parms;
  cell_config_parms.library_path = getenv("ALF_SPU_LIB_PATH");

  config_parms = &cell_config_parms;
  if (sizeof(void *) == 4)
    sprintf(library_name, "alf_matrix_add_step4_cell_spu.so");
  else
    sprintf(library_name, "alf_matrix_add_step4_cell_spu64.so");
#endif

  rc = alf_init(config_parms, &alf_handle);
  if (rc < 0) {
    fprintf(stderr, "alf init failed\n");
    return 1;
  }

  rc = alf_query_system_info(alf_handle, ALF_QUERY_NUM_ACCEL, 0, &nodes);
  if (rc < 0) {
    fprintf(stderr, "Failed to call alf_query_system_info.\n");
    return 1;
  } else if (nodes <= 0) {
    fprintf(stderr, "Cannot allocate spe to use.\n");
    return 1;
  }

  rc = alf_num_instances_set(alf_handle, 1);
  if (rc < 0) {
    fprintf(stderr, "Cannot init ALF library(%d).\n", rc);
    return rc;
  }

  alf_task_desc_create(alf_handle, 0,&task_desc_handle);

  alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_WB_PARM_CTX_BUF_SIZE, sizeof(add_parms_t));
  alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_WB_IN_BUF_SIZE, H * V * sizeof(float)); // 32k
  alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_WB_OUT_BUF_SIZE, 0);    // 0k
  alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_WB_INOUT_BUF_SIZE,  H * V * sizeof(float));
  alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_NUM_DTL_ENTRIES, 32);
  alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_TSK_CTX_SIZE, 0);
  alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_MAX_STACK_SIZE, 8192);
  alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_PARTITION_ON_ACCEL, is_spu_dma);
  alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_IMAGE_REF_L, (unsigned long long)spu_image_name);
  alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_LIBRARY_REF_L, (unsigned long long)library_name);
  alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_KERNEL_REF_L, (unsigned long long)kernel_name);
  alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_INPUT_DTL_REF_L, (unsigned long long)input_dtl_name);
  alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_OUTPUT_DTL_REF_L, (unsigned long long)output_dtl_name);
  alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_CTX_SETUP_REF_L, (unsigned long long)ctx_setup_name);
  alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_CTX_MERGE_REF_L, (unsigned long long)ctx_merge_name);

  /* creating task */
  rc = alf_task_create(task_desc_handle, NULL, 1, 0, 0, &task_handle);
  if (rc < 0) {
    fprintf(stderr, "Cannot create ALF task(%d).\n", rc);
    alf_exit(alf_handle, ALF_EXIT_POLICY_FORCE, 0);
    return 1;
```

```
  }

#ifdef _ALF_PLATFORM_HYBRID_
  parm.h = bswap_32(H);
  parm.v = bswap_32(V);
  parm.size = bswap_32(H * V);
#else
  parm.h = H;
  parm.v = V;
  parm.size = H * V;
#endif
  rc = alf_dataset_create(alf_handle, &dataset_handle);
  if (rc < 0) {
    fprintf(stderr, "Call alf_dataset_create error: %d\n", rc);
    return 1;
  }
  rc = alf_dataset_buffer_add(dataset_handle, mat_a, NUM_ROW*NUM_COL*sizeof(float), ALF_DATASET_READ_ONLY);
  if (rc < 0) {
    fprintf(stderr, "Call alf_dataset_buffer_add error: %d\n", rc);
    return 1;
  }
  rc = alf_dataset_buffer_add(dataset_handle, mat_b, NUM_ROW*NUM_COL*sizeof(float), ALF_DATASET_READ_WRITE);
  if (rc < 0) {
    fprintf(stderr, "Call alf_dataset_buffer_add error: %d\n", rc);
    return 1;
  }
  rc = alf_task_dataset_associate(task_handle, dataset_handle);
  if (rc < 0) {
    fprintf(stderr, "Call alf_task_dataset_associate error: %d\n", rc);
    return 1;
  }
  /* creating wb and adding param & io buffer */
  for (i = 0; i < NUM_ROW; i += H) {

    alf_wb_create(task_handle, ALF_WB_SINGLE, 0, &wb_handle);

    if (!is_spu_dma) {
      alf_wb_dtl_begin(wb_handle, ALF_BUF_IN, 0);
      alf_wb_dtl_entry_add(wb_handle, &mat_a[i][0], H * V, ALF_DATA_FLOAT);
      alf_wb_dtl_end(wb_handle);

      alf_wb_dtl_begin(wb_handle, ALF_BUF_OVL_INOUT, 0);
      if (alf_wb_dtl_entry_add(wb_handle, &mat_b[i][0], H * V, ALF_DATA_FLOAT) != 0)
      {
        printf ("APP ERROR***\n");

      }
      alf_wb_dtl_end(wb_handle);

      parm.in_data = 0;
      parm.out_data = 0;
    } else {
#ifdef _ALF_PLATFORM_HYBRID_
      //printf ("Matrix addtion, in_data = 0x%llx\n", (unsigned long long)(&mat_a[i][0]));
      //printf ("Matrix addtion, out_data = 0x%llx\n", (unsigned long long)(&mat_b[i][0]));

      parm.in_data = bswap_64((unsigned long long)&mat_a[i][0]);
      parm.out_data = bswap_64((unsigned long long)&mat_b[i][0]);
#else
      parm.in_data = (unsigned int) &mat_a[i][0];
      parm.out_data = (unsigned int) &mat_b[i][0];
#endif
    }

    alf_wb_parm_add(wb_handle, (void *) (&parm), sizeof(parm), ALF_DATA_BYTE, 0);
    alf_wb_enqueue(wb_handle);
  }

  alf_task_finalize(task_handle);

  /* alf task wait until wb processed */
  alf_task_wait(task_handle, -1);
  alf_task_destroy(task_handle);
  /* end of alf routine */

  alf_exit(alf_handle, ALF_EXIT_POLICY_WAIT, -1);

  printf("Matrix Addition computed. Verifying results...\n");
```

```
  /* verifying */
  for (i = 0; i < NUM_ROW; i++) {
    for (j = 0; j < NUM_COL; j++) {
      if (mat_b[i][j] != (float) ((i * NUM_COL + j) * 3)) {
        printf("Matrix Addition failed verification. b[%d][%d] != %f\n",i,j,(float) ((i * NUM_COL + j) * 3));
        return -1;
      }
    }
  }

  printf("Matrix Addition successfully verified.\n");

  return 0;
}
```

Appendix B. ALF trace events

The following shows the ALF trace events that are defined. In general, there are two trace hooks per API:

- The first traces the input parameters
- The second traces the output values as well as the time interval of the API call

ALF API hooks

Enabled with: TRACE_ALF_DEBUG.

Table 1. ALF debug hooks

Hook identifier	Traced values
_ALF_DATASET_ASSOCIATE_ENTRY	task_handle, dataset_handle
_ALF_DATASET_ASSOCIATE_EXIT_INTERVAL	retcode
_ALF_DATASET_BUFFER_ADD_ENTRY	dataset_handle, buffer, size, access_mode
_ALF_DATASET_BUFFER_ADD_EXIT_INTERVAL	retcode
_ALF_DATASET_CREATE_ENTRY	alf_handle, p_dataset_handle
_ALF_DATASET_CREATE_EXIT_INTERVAL	dataset_handle, retcode
_ALF_DATASET_DESTROY_ENTRY	dataset_handle
_ALF_DATASET_DESTROY_EXIT_INTERVAL	retcode
_ALF_EXIT_ENTRY	alf_handle, exit_policy, timeout
_ALF_EXIT_EXIT_INTERVAL	retcode
_ALF_GENERIC_DEBUG	long1, long2, long3, long4, long5, long6, long7, long8, long9, long10
_ALF_INIT_ENTRY	sys_config_info, alf_handle_ptr
_ALF_INIT_EXIT_INTERVAL	rtn
_ALF_NUM_INSTANCES_SET_ENTRY	alf_handle, number_of_instances
_ALF_NUM_INSTANCES_SET_EXIT_INTERVAL	retcode
_ALF_QUERY_SYSINFO_ENTRY	alf_handle, query_info, accel_type, p_query_result
_ALF_QUERY_SYSINFO_EXIT_INTERVAL	query_result, retcode
_ALF_REGISTER_ERROR_HANDLER_ENTRY	alf_handle, error_handler_function, p_context
_ALF_REGISTER_ERROR_HANDLER_EXIT_INTERVAL	retcode
_ALF_TASK_CREATE_ENTRY	task_desc_handle, p_task_context_data, num_accelerators, tsk_attr, wb_dist_size, p_task_handle
_ALF_TASK_CREATE_EXIT_INTERVAL	task_handle, retcode
_ALF_TASK_DEPENDS_ON_ENTRY	task_handle_dependent, task_handle
_ALF_TASK_DEPENDS_ON_EXIT_INTERVAL	retcode
_ALF_TASK_DESC_CREATE_ENTRY	alf_handle, accel_type, task_desc_handle_ptr
_ALF_TASK_DESC_CREATE_EXIT_INTERVAL	desc_info_handle, retcode
_ALF_TASK_DESC_CTX_ENTRY_ADD_ENTRY	task_desc_handle, data_type, size
_ALF_TASK_DESC_CTX_ENTRY_ADD_EXIT_INTERVAL	retcode

Table 1. ALF debug hooks (continued)

Hook identifier	Traced values
_ALF_TASK_DESC_DESTROY_ENTRY	task_desc_handle
_ALF_TASK_DESC_DESTROY_EXIT_INTERVAL	retcode
_ALF_TASK_DESC_SET_INT32_ENTRY	task_desc_handle, field, value
_ALF_TASK_DESC_SET_INT32_EXIT_INTERVAL	retcode
_ALF_TASK_DESC_SET_INT64_ENTRY	task_desc_handle, field, value
_ALF_TASK_DESC_SET_INT64_EXIT_INTERVAL	retcode
_ALF_TASK_DESTROY_ENTRY	task_handle
_ALF_TASK_DESTROY_EXIT_INTERVAL	retcode
_ALF_TASK_EVENT_HANDLER_REGISTER_ENTRY	task_handle, task_event_handler, p_data, data_size, event_mask
_ALF_TASK_EVENT_HANDLER_REGISTER_EXIT_INTERVAL	retcode
_ALF_TASK_FINALIZE_ENTRY	task_handle
_ALF_TASK_FINALIZE_EXIT_INTERVAL	retcode
_ALF_TASK_QUERY_ENTRY	talk_handle, p_unfinished_wbs, p_total_wbs
_ALF_TASK_QUERY_EXIT_INTERVAL	unfinished_wbs, total_wbs, retcode
_ALF_TASK_WAIT_ENTRY	task_handle, time_out
_ALF_TASK_WAIT_EXIT_INTERVAL	retcode
_ALF_WB_CREATE_ENTRY	task_handle, work_block_type, repeat_count, p_wb_handle
_ALF_WB_CREATE_EXIT_INTERVAL	wb_handle, retcode
_ALF_WB_DTL_SET_BEGIN_ENTRY	wb_handle, buffer_type, offset_to_the_local_buffer
_ALF_WB_DTL_SET_BEGIN_EXIT_INTERVAL	retcode
_ALF_WB_DTL_SET_END_ENTRY	wb_handle
_ALF_WB_DTL_SET_END_EXIT_INTERVAL	retcode
_ALF_WB_DTL_SET_ENTRY_ADD_ENTRY	wb_handle, p_address, size_of_data, data_type
_ALF_WB_DTL_SET_ENTRY_ADD_EXIT_INTERVAL	retcode
_ALF_WB_ENQUEUE_ENTRY	wb_handle
_ALF_WB_ENQUEUE_EXIT_INTERVAL	retcode
_ALF_WB_PARM_ADD_ENTRY	wb_handle, pdata, size_of_data, data_type, address_alignment
_ALF_WB_PARM_ADD_EXIT_INTERVAL	retcode

ALF performance hooks

These trace hooks are enabled by LIBALF_PERF group (0x08) in the config file.

The COUNTERS and TIMERS hooks contain data that is accumulated during the ALF calls. Currently, that data and these trace events will get reported at various ALF exit calls.

Table 2. ALF performance hooks

Hook Identifier	Traced values
_ALF_GENERIC_PERFORM_HOST	long1, long2, long3, long4, long5, long6, long7, long8, long9, long10
_ALF_GENERIC_PERFORM_SPU	long1, long2, long3, long4, long5, long6, long7, long8, long9, long10
_ALF_HOST_COUNTERS	alf_task_creates, alf_task_waits, alf_wb_enqueues, thread_total_count, thread_reuse_count, x
_ALF_HOST_TIMERS	alf_runtime, alf_accel_utilize, x1, x2
_ALF_SPU_COUNTERS	alf_input_bytes, alf_output_bytes, alf_workblock_total, double_buffer_used, x1, x2
_ALF_SPU_TIMERS	alf_lqueue_empty, alf_wait_data_dtl, alf_prep_input_dtl, alf_prep_output_dtl, alf_compute_kernel, alf_spu_task_run, x1, x2
_ALF_TASK_BEFORE_EXEC_INTERVAL	task_flag
_ALF_TASK_CONTEXT_MERGE_INTERVAL	task_flag
_ALF_TASK_CONTEXT_SWAP_INTERVAL	task_flag
_ALF_TASK_EXEC_INTERVAL	task_flag
_ALF_THREAD_RUN_INTERVAL	task_flag
_ALF_WAIT_FIRST_WB_INTERVAL	task_flag, wb_flag, packet_flag
_ALF_WB_COMPUTE_KERNEL_INTERVAL	task_flag, wb_flag, wb idx
_ALF_WB_DATA_TRANSFER_WAIT_INTERVAL	task_flag, wb_flag, wb idx
_ALF_WB_DTL_PREPARE_IN_INTERVAL	task_flag, wb_flag, wb idx
_ALF_WB_DTL_PREPARE_OUT_INTERVAL	task_flag, wb_flag, wb idx
_ALF_WB_LQUEUE_EMPTY_INTERVAL	task_flag, packet_flag

ALF SPU hooks

These trace hooks are enabled by LIBALF_SPU group (0x09) in the config file.

Table 3. ALF SPU hooks

Hook identifier	Traced values
_ALF_ACCEL_COMP_KERNEL_ENTRY	p_task_context, p_parm_ctx_buffer, p_input_buffer, p_output_buffer, p_inout_buffer, current_iter, num_iter
_ALF_ACCEL_COMP_KERNEL_EXIT	retcode
_ALF_ACCEL_DTL_BEGIN_ENTRY	p_dtl, buf_type, offset
_ALF_ACCEL_DTL_BEGIN_EXIT	p_dtl, retcode
_ALF_ACCEL_DTL_END_ENTRY	p_dtl
_ALF_ACCEL_DTL_END_EXIT _	retcode
_ALF_ACCEL_DTL_ENTRY_ADD_ENTRY	p_dtl, data_size, data_type, p_host_address
_ALF_ACCEL_DTL_ENTRY_ADD_EXIT	retcode
_ALF_ACCEL_INPUT_DTL_PREPARE_ENTRY	p_task_context, p_parm_ctx_buffer, p_dtl, current_iter, num_iter
_ALF_ACCEL_INPUT_DTL_PREPARE_EXIT	retcode
_ALF_ACCEL_NUM_INSTANCES	retcode

Table 3. ALF SPU hooks (continued)

Hook identifier	Traced values
_ALF_ACCEL_OUTPUT_DTL_PREPARE_ENTRY	p_task_context, p_parm_ctx_buffer, p_io_container, current_iter, num_iter
_ALF_ACCEL_OUTPUT_DTL_PREPARE_EXIT	retcode
_ALF_ACCEL_TASK_CONTEXT_MERGE_ENTRY	p_task_context_to_be_merged, p_task_context
_ALF_ACCEL_TASK_CONTEXT_MERGE_EXIT	retcode
_ALF_ACCEL_TASK_CONTEXT_SETUP_ENTRY	p_task_context
_ALF_ACCEL_TASK_CONTEXT_SETUP_EXIT	retcode
_ALF_INSTANCES_ID	retcode
_ALF_SPE_GENERIC_DEBUG	long1, long2, long3, long4, long5, long6, long7, long8, long9, long10

Appendix C. Attributes and descriptions

The following table is a list of attributes.

Table 4. Attributes and descriptions

Attribute name	Description
ALF_QUERY_NUM_ACCEL	Return the number of accelerators of a particular type accel_type in the system.
ALF_QUERY_HOST_MEM_SIZE	Return the memory size of control nodes up to 4T bytes, in units of kilo bytes (2^10 bytes).
ALF_QUERY_HOST_MEM_SIZE_EXT	Return the memory size of control nodes, in units of 4T bytes (2^42 bytes)
ALF_QUERY_ACCEL_MEM_SIZE	Return the memory size of accelerator nodes up to 4T bytes, in units of kilo bytes (2^10 bytes).
ALF_QUERY_ACCEL_MEM_SIZE_EXT	Return the memory size of accelerator nodes, in units of 4T bytes (2^42 bytes).
ALF_QUERY_HOST_ADDR_ALIGN	Return the basic requirement of memory address alignment on control node side, in exponential of 2.
ALF_QUERY_ACCEL_ADDR_ALIGN	Return the basic requirement of memory address alignment on accelerator node side, in exponential of 2.
ALF_QUERY_DTL_ADDR_ALIGN	Return the address alignment of data transfer list entries, in exponential of 2.
ALF_ACCEL_TYPE_SPE	Accelerator type.
ALF_EXIT_POLICY_FORCE	Perform a shutdown immediately and aborts all unfinished tasks if there are any.
ALF_EXIT_POLICY_WAIT	Wait for all tasks to be processed and then shuts down.
ALF_EXIT_POLICY_TRY	Return with a failure if there are unfinished tasks.
ALF_TASK_DESC_WB_PARM_CTX_BUF_SIZE	Size of the work block parameter buffer.
ALF_TASK_DESC_WB_IN_BUF_SIZE	Size of the work block input buffer.
ALF_TASK_DESC_WB_OUT_BUF_SIZE	Size of the work block output buffer.
ALF_TASK_DESC_WB_INOUT_BUF_SIZE	Size of the work block overlapped input/output buffer.
ALF_TASK_DESC_NUM_DTL_ENTRIES	Maximum number of entries for the data transfer list.
ALF_TASK_DESC_TSK_CTX_SIZE	Size of the task context buffer.
ALF_TASK_DESC_PARTITION_ON_ACCEL	Specifies whether the accelerator functions are invoked to generate data transfer lists for input and output data.
ALF_TASK_DESC_MAX_STACK_SIZE	Specify the maximum stack size.
ALF_TASK_DESC_ACCEL_LIBRARY_REF_L	Specify the name of the library that the accelerator image is contained in
ALF_TASK_DESC_ACCEL_IMAGE_REF_L	Specify the name of the accelerator image that's contained in the library.

Table 4. Attributes and descriptions (continued)

Attribute name	Description
ALF_TASK_DESC_ACCEL_KERNEL_REF_L	Specify the name of the computational kernel function, this usually is a string constant that the accelerator runtime could use to find the correspondent function.
ALF_TASK_DESC_ACCEL_INPUT_DTL_REF_L	Specify the name of the input list prepare function, this usually is a string constant that the accelerator runtime could use to find the correspondent function.
ALF_TASK_DESC_ACCEL_OUTPUT_DTL_REF_L	Specify the name of the output list prepare function, this usually is a string constant that the accelerator runtime could use to find the correspondent function.
ALF_TASK_DESC_ACCEL_CTX_SETUP_REF_L	Specify the name of the context setup function, this usually is a string constant that the accelerator runtime could use to find the correspondent function.
ALF_TASK_DESC_ACCEL_CTX_MERGE_REF_L	Specify the name of the context merge function, this usually a string constant that the accelerator runtime could use to find the correspondent function.
ALF_TASK_ATTR_SCHED_FIXED	The task must be scheduled on `num_instances` of accelerators.
ALF_TASK_ATTR_WB_CYCLIC	The work blocks for this task is distributed to the accelerators in a cyclic order as specified by `num_accelerators`.
ALF_TASK_EVENT_TYPE_T	Defined as followed: • ALF_TASK_EVENT_FINALIZED: This task has been finalized. No additional work block can be added to this task. • ALF_TASK_EVENT_READY: This task has been scheduled for execution. • ALF_TASK_EVENT_FINISHED: All work blocks in this task have been processed. • ALF_TASK_EVENT_INSTANCE_START: One new instance of the task is started on an accelerator after the event handler returns • ALF_TASK_EVENT_INSTANCE_END: One existing instance of the task ends and the task context has been copied out to the original location or has been merged to another current instance of the same task. • ALF_TASK_EVENT_DESTROY: The task is destroyed explicitly
ALF_WB_SINGLE	Create a single use work block.
ALF_WB_MULTI (Level 1)	Create a multi use work block. This work block type is only supported when the task is created with ALF_PARTITION_ON_ACCELERATOR.
ALF_BUF_IN	Input to the input-only buffer.
ALF_BUF_OUT	Output from the output only buffer.
ALF_BUF_OVL_IN	Input to the overlapped buffer.
ALF_BUF_OVL_OUT	Output from the overlapped buffer.

Table 4. Attributes and descriptions (continued)

Attribute name	Description
ALF_BUF_OVL_INOUT	In/out to/from the overlapped buffer.
ALF_DATASET_READ_ONLY	The dataset is read-only. Work blocks referencing the data in this buffer cannot update this buffer as an output buffer.
ALF_DATASET_WRITE_ONLY	The dataset is write-only. Work blocks referencing the data in this buffer as input data results in indeterminate behavior.
ALF_DATASET_READ_WRITE	The dataset allows both read and write access. Work blocks can use this buffer as input buffers and output buffers and/or inout buffers.

Appendix D. Error codes and descriptions

The following table is a list of the ALF error codes.

Table 5. Error codes and descriptions

Error	Error code	Description
ALF_ERR_PERM	1	No permission
ALF_ERR_SRCH	3	No such task
ALF_ERR_2BIG	7	I/O request out of scope
ALF_ERR_NOEXEC	8	Runtime error
ALF_ERR_BADF	9	Bad handle
ALF_ERR_AGAIN	11	Try again
ALF_ERR_NOMEM	12	Out of memory
ALF_ERR_FAULT	14	Invalid address
ALF_ERR_BUSY	16	Resource busy
ALF_ERR_INVAL	22	Invalid argument
ALF_ERR_RANGE	34	Numerical results or `args` out of valid range
ALF_ERR_NOSYS	38	Function or features not implemented
ALF_ERR_BADR	53	The resource request cannot be fulfilled
ALF_ERR_NODATA	61	No more data available
ALF_ERR_TIME	62	Timeout
ALF_ERR_COMM	70	Generic communication error
ALF_ERR_PROTO	71	Internal protocol error
ALFF_ERR_BADMSG	74	Internal protocol error
ALF_ERR_OVERFLOW	75	Value out of range when converting
ALF_ERR_NOBUFS	105	No buffer space available
ALF_ERR_GENERIC	1000	Generic ALF internal error
ALF_ERR_ACCEL	2000	Generic accelerator error

Appendix E. Accessibility features

Accessibility features help users who have a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

The following list includes the major accessibility features:

- Keyboard-only operation
- Interfaces that are commonly used by screen readers
- Keys that are tactilely discernible and do not activate just by touching them
- Industry-standard devices for ports and connectors
- The attachment of alternative input and output devices

IBM® and accessibility

See the IBM Accessibility Center at http://www.ibm.com/able/ for more information about the commitment that IBM has to accessibility.

Notices

This information was developed for products and services offered in the U.S.A.

The manufacturer may not offer the products, services, or features discussed in this document in other countries. Consult the manufacturer's representative for information on the products and services currently available in your area. Any reference to the manufacturer's product, program, or service is not intended to state or imply that only that product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any intellectual property right of the manufacturer may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any product, program, or service.

The manufacturer may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the manufacturer.

For license inquiries regarding double-byte (DBCS) information, contact the Intellectual Property Department in your country or send inquiries, in writing, to the manufacturer.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: THIS INFORMATION IS PROVIDED "AS IS " WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. The manufacturer may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to Web sites not owned by the manufacturer are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this product and use of those Web sites is at your own risk.

The manufacturer may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact the manufacturer.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning products not produced by this manufacturer was obtained from the suppliers of those products, their published announcements or other publicly available sources. This manufacturer has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to products not produced by this manufacturer. Questions on the capabilities of products not produced by this manufacturer should be addressed to the suppliers of those products.

All statements regarding the manufacturer's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

The manufacturer's prices shown are the manufacturer's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to the manufacturer, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. The manufacturer, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

CODE LICENSE AND DISCLAIMER INFORMATION:

The manufacturer grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, THE MANUFACTURER, ITS PROGRAM DEVELOPERS AND SUPPLIERS, MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS THE MANUFACTURER, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information in softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM
developerWorks
PowerPC
PowerPC® Architecture
Resource Link

Adobe, Acrobat, Portable Document Format (PDF), and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Cell Broadband Engine™ and Cell/B.E. are trademarks of Sony Computer Entertainment, Inc., in the United States, other countries, or both and is used under license therefrom.

Linux® is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of the manufacturer.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of the manufacturer.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any data, software or other intellectual property contained therein.

The manufacturer reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by the manufacturer, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

THE MANUFACTURER MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THESE PUBLICATIONS ARE PROVIDED ″AS-IS″ AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Related documentation

This topic helps you find related information.

Document location

Links to documentation for the SDK are provided on the developerWorks® Web site located at:

`http://www-128.ibm.com/developerworks/power/cell/`

Click on the **Docs** tab.

The following documents are available, organized by category:

Architecture

- *Cell Broadband Engine Architecture*
- *Cell Broadband Engine Registers*
- *SPU Instruction Set Architecture*

Standards

- *C/C++ Language Extensions for Cell Broadband Engine Architecture*
- *SPU Assembly Language Specification*
- *SPU Application Binary Interface Specification*
- *SIMD Math Library Specification for Cell Broadband Engine Architecture*
- *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification*

Programming

- *Cell Broadband Engine Programming Handbook*
- *Programming Tutorial*
- *SDK for Multicore Acceleration Version 3.0 Programmer's Guide*

Library

- *SPE Runtime Management library*
- *SPE Runtime Management library Version 1.2 to Version 2.0 Migration Guide*
- *Accelerated Library Framework for Cell Programmer's Guide and API Reference*
- *Accelerated Library Framework for Hybrid-x86 Programmer's Guide and API Reference*
- *Data Communication and Synchronization for Cell Programmer's Guide and API Reference*
- *Data Communication and Synchronization for Hybrid-x86 Programmer's Guide and API Reference*
- *SIMD Math Library Specification*
- *Monte Carlo Library API Reference Manual (Prototype)*

Installation

- *SDK for Multicore Acceleration Version 3.0 Installation Guide*

IBM XL C/C++ Compiler and IBM XL Fortran Compiler

Details about documentation for the compilers is available on the developerWorks Web site.

IBM Full-System Simulator and debugging documentation

Detail about documentation for the simulator and debugging tools is available on the developerWorks Web site.

PowerPC Base

- *PowerPC Architecture™ Book, Version 2.02*
  - *Book I: PowerPC User Instruction Set Architecture*
  - *Book II: PowerPC Virtual Environment Architecture*
  - *Book III: PowerPC Operating Environment Architecture*
- *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual Version 2.07c*

Glossary

ABI

Application Binary Interface. This is the standard that a program follows to ensure that code generated by different compilers (and perhaps linking with various, third-party libraries) run correctly on the Cell BE. The ABI defines data types, register use, calling conventions and object formats.

accelerator

General or special purpose processing element in a hybrid system. An accelerator can have a multi-level architecture with both host elements and accelerator elements. An accelerator, as defined here, is a hierarchy with potentially multiple layers of hosts and accelerators. An accelerator element is always associated with one host. Aside from its direct host, an accelerator cannot communicate with other processing elements in the system. The memory subsystem of the accelerator can be viewed as distinct and independent from a host. This is referred to as the subordinate in a cluster collective.

ALF

Accelerated Library Framework. This an API that provides a set of services to help programmers solving data parallel problems on a hybrid system. ALF supports the multiple-program-multiple-data (MPMD) programming style where multiple programs can be scheduled to run on multiple accelerator elements at the same time. ALF offers programmers an interface to partition data across a set of parallel processes without requiring architecturally-dependent code.

API

Application Program Interface.

ATO

Atomic Unit. Part of an SPE's MFC. It is used to synchronize with other processor units.

Broadband Engine

See *CBEA*.

C++

C++ is an object-orientated programming language, derived from C.

cache

High-speed memory close to a processor. A cache usually contains recently-accessed data or instructions, but certain cache-control instructions can lock, evict, or otherwise modify the caching of data or instructions.

CBEA

Cell Broadband Engine Architecture. A new architecture that extends the 64-bit PowerPC Architecture. The CBEA and the Cell Broadband Engine are the result of a collaboration between Sony, Toshiba, and IBM, known as STI, formally started in early 2001.

Cell BE processor

The Cell BE processor is a multi-core broadband processor based on IBM's Power Architecture.

Cell Broadband Engine processor

See *Cell BE*.

cluster

A collection of nodes.

compiler

A programme that translates a high-level programming language, such as C++, into executable code.

computational kernel

Part of the accelerator code that does stateless computation task on one piece of input data and generates corresponding output results.

compute task

An accelerator execution image that consists of a compute kernel linked with the accelerated library framework accelerator runtime library.

data set

An ALF data set is a logical set of data buffers.

DMA

Direct Memory Access. A technique for using a special-purpose controller to generate the source and destination addresses for a memory or I/O transfer.

DMA command

A type of MFC command that transfers or controls the transfer of a memory location containing data or instructions. See *MFC*.

DMA list

A sequence of transfer elements (or list entries) that, together with an initiating DMA-list command, specify a sequence of DMA transfers between a single area of LS and discontinuous areas in main storage. Such lists are stored in an SPE's LS, and the sequence of transfers is initiated with a DMA-list command such as `getl` or `putl`. DMA-list commands can only be issued by programs running on an SPE, but the PPE or other devices can create and store the lists in an SPE's LS. DMA lists can be used to implement scatter-gather functions between main storage and the LS.

DMA-list command

A type of MFC command that initiates a sequence of DMA transfers specified by a DMA list stored in an SPE's LS. See *DMA list*.

EA

See *Effective address*.

effective address

An address generated or used by a program to reference memory. A memory-management unit translates an effective address (EA) to a virtual address (VA), which it then translates to a real address (RA) that accesses real (physical) memory. The maximum size of the effective address space is 2^{64} bytes.

exception

An error, unusual condition, or external signal that may alter a status bit and will cause a corresponding interrupt, if the interrupt is enabled. See interrupt.

FFT

Fast Fourier Transform.

GCC

GNU C compiler

handle

A handle is an abstraction of a data object; usually a pointer to a structure.

host

A general purpose processing element in a hybrid system. A host can have multiple accelerators attached to it. This is often referred to as the master node in a cluster collective.

HTTP

Hypertext Transfer Protocol. A method used to transfer or convey information on the World Wide Web.

Hybrid

A module comprised of two Cell BE cards connected via an AMD Opteron processor.

IDL

Interface definition language. Not the same as CORBA IDL

kernel

The core of an operating which provides services for other parts of the operating system and provides multitasking. In Linux or UNIX operating system, the kernel can easily be rebuilt to incorporate enhancements which then become operating-system wide.

latency

The time between when a function (or instruction) is called and when it returns. Programmers often optimize code so that functions return as quickly as possible; this is referred to as the low-latency approach to optimization. Low-latency designs often leave the processor data-starved, and performance can suffer.

local store

The 256-KB local store associated with each SPE. It holds both instructions and data.

LS

See *local store*.

main storage

The effective-address (EA) space. It consists physically of real memory (whatever is external to the memory-interface controller, including both volatile and nonvolatile memory), SPU LSs, memory-mapped registers and arrays, memory-mapped I/O devices (all I/O is memory-mapped), and pages of virtual memory that reside on disk. It does not include caches or execution-unit register files. See also *local store*.

main thread

The main thread of the application. In many cases, Cell BE architecture programs are multi-threaded using multiple SPEs running concurrently. A typical scenario is that the application consists of a main thread that creates as many SPE threads as needed and the application organizes them.

MFC

Memory Flow Controller. Part of an SPE which provides two main functions: it moves data via DMA between the SPE's local store (LS) and main storage, and it synchronizes the SPU with the rest of the processing units in the system.

MPMD

Multiple Program Multiple Data. Parallel programming model with several distinct executable programs operating on different sets of data.

node

A node is a functional unit in the system topology, consisting of one host together with all the accelerators connected as children in the topology (this includes any children of accelerators).

PDF

Portable document format.

pipelining

A technique that breaks operations, such as instruction processing or bus transactions, into smaller stages so that a subsequent stage in the pipeline can begin before the previous stage has completed.

PPE

PowerPC Processor Element. The general-purpose processor in the Cell.

PPE

PowerPC Processor Element. The general-purpose processor in the Cell BE processor.

PPU

PowerPC Processor Unit. The part of the *PPE* that includes the execution units, memory-management unit, and L1 cache.

process

A process is a standard UNIX-type process with a separate address space.

program section

See *code section*.

SDK

Software development toolkit for Multicore Acceleration. A complete package of tools for application development.

section

See *code section*.

SIMD

Single Instruction Multiple Data. Processing in which a single instruction operates on multiple data elements that make up a vector data-type. Also known as vector processing. This style of programming implements data-level parallelism.

SPE

Synergistic Processor Element. Extends the PowerPC 64 architecture by acting as cooperative offload processors (synergistic processors), with the direct memory access (DMA) and synchronization mechanisms to communicate with them (memory flow control), and with enhancements for real-time management. There are 8 SPEs on each cell processor.

SPU

Synergistic Processor Unit. The part of an SPE that executes instructions from its local store (LS).

SPU

Synergistic Processor Unit. The part of an SPE that executes instructions from its local store (LS).

synchronization

The order in which storage accesses are performed.

thread

A sequence of instructions executed within the global context (shared memory space and other global resources) of a process that has created (spawned) the thread. Multiple threads (including multiple instances of the same sequence of instructions) can run simultaneously if each thread has its own architectural state (registers, program counter, flags, and other program-visible state). Each SPE can support only a single thread at any one time. Multiple SPEs can simultaneously support multiple threads. The PPE supports two threads at any one time, without the need for software to create the threads. It does this by duplicating the architectural state. A thread is typically created by the `pthreads` library.

vector

An instruction operand containing a set of data elements packed into a one-dimensional array. The elements can be fixed-point or floating-point values. Most Vector/SIMD Multimedia Extension and SPU SIMD instructions operate on vector operands. Vectors are also called SIMD operands or packed operands.

virtual memory

The address space created using the memory management facilities of a processor.

virtual storage

See *virtual memory*.

work block

A basic unit of data to be managed by the framework. It consists of one piece of the partitioned data, the corresponding output buffer, and related parameters. A work block is associated with a task. A task can have as many work blocks as necessary.

workload

A set of code samples in the SDK that characterizes the performance of the architecture, algorithms, libraries, tools, and compilers.

work queue

An internal data structure of the accelerated library framework that holds the lists of work blocks to be processed by the active instances of the compute task.

x86

Generic name for Intel-based processors.

XLC

The IBM optimizing C/C++ compiler.

Index

Printed in USA

SC33-8406-00