Psi Lambda LLC ψ(λκ)

Kappa Library

User Guide

(Parallel computing made easier)
version 1.5

Table of Contents

Introduction

Please read Chapters 1 and 2 of the NVIDIA CUDA C Programming Guide prior to reading this or other Kappa manuals. It is recommended that you read all of the NVIDIA CUDA C Programming Guide prior to using Kappa. Kappa uses only the CUDA driver API that is installed by the NVIDIA driver (and toolkit) installation. It has no dependencies on but will interoperate with the CUDA runtime libraries

Kappa provides a library for creating processes to use combinations of CPUs and a GPU for tasks. Within a single host program process, a Kappa process can be created for each CUDA GPU—using all GPUs. Each Kappa process can use all of the multiprocessors of each GPU, share all of the CPUs of the host system, have its own separate namespace, and have its own separate CUDA context. The Kappa library provides:

- access to the GPU and instantiated kernel properties at runtime
- nvcc and CUDA JIT compilation loading
- fully concurrent C, C++ OpenMP, and CUDA kernel execution
- dynamic sizing of data and kernel invocation
- dynamic scheduling and cancellation of the execution of related steps
- process level functional blocks (named or anonymous subroutines and named functions) built from dynamically scheduled execution of mixtures of C++ and CUDA kernels
- subject domain extensibility

The Kappa library was designed to be especially ease to use by providing, for example:

- interactive or batch capability for rapid development or testing,
- resource management (constructor, destructor, and automatic data copy),
- fail-safe default behavior,
- thread-safe locking,
- and exception handling.

Overview

Most people are familiar with procedural programming which is also known as imperative programming. This consists of specifying the steps, in order, that a program must take to reach the desired goal. This form of programming does not work well with parallel computing hardware such as multiple CPUs or GPUs. These types of hardware execute calculations simultaneously (in parallel). Instead of a "function" that specifies the steps, in order, that a program executes, parallel computing has kernels that take a set of data and simultaneously, in parallel, executes a series of calculations across the set of data. These calculations can either change the set of data where it was originally stored (in memory) and/or can write the results to some other (memory) storage. (One other difference is that kernels do not return a single value but instead return data by modifying one or more of the passed arguments.)

Kappa needs to keep track of the relationships for how data flows through a process so that it can be sure to schedule steps in the correct order. The arguments for C++ and CUDA functions and kernels do not specify whether a given argument is used for input, for output, or for both input and output. If the argument is a literal value such as a number or a string, then it has to be an input, but in other cases, specifically if the argument is a pointer or reference, there is no information about the arguments use and therefore no way to track the relationships between arguments of kernels and the possible flow flow of data.

This method for tracking data for parallel computation is usually called Producer/Consumer resource tracking. The data objects are considered resources and the processing steps are considered as producers and/or consumers of these resources. The idea is that each step in a process is either a producer of a particular resource, a consumer of a particular resource, or both a consumer and a producer if it takes a resource and modifies it.

So that Kappa can track the relationships of arguments and ensure the proper flow of data in scheduled for processing steps, Kappa introduces a new notation. To help understand this notation, examples will be discussed that refer to some hypothetical kernel, K, and data sets A, B, and C. These are data sets should be thought of as storage locations such as a filename or a memory storage address—the data at the names of those locations may be changed but the name of the location (filename) does not.

Here are the examples:

A is the argument to K. The data set A is transformed, in place, by the kernel, K.

A, B, and C are the arguments to K. The data set C is written to by the kernel, K, using A and B as inputs.

A, B, and C are the arguments to K. The data set C is written to by the kernel K, using A, B, and C as inputs.

A, B, and C are the arguments to K. The data sets A, B, and C are written to by the kernel K.

Kappa will use the following notation to show these relationships:

K(A) [A = A] or K(A) [A]
K(A,B,C) [C = A B]
K(A,B,C) [C = A B C]

```
    K(A,B,C) [ A B C = ]
```

where, within the parenthesis, other arguments that are not relevant to the proper execution order are also sometimes given:

```
    K(A,4,5) [A]
    K(A,B, 42, “a string of text”) [ C = A B ]
    K(A,B,C, (void *)cpp_class, 50) [ C = A B C ]
```

and so on.

This notation provides the information needed to schedule calculations in the proper order—to define a process to accomplish (one or more) tasks. For example, statements like:

```
    Variable A;
    kernel1 (A) [A];
    kernel2 (A,B) [ B = A ];
```

give a flow of the data of:

```
    A -> A -> B
```

where the data set A is declared (produced), kernel1 transforms data set A, in place (takes the data in A, changes it, and writes it back to data set A--consumes A and produces a new A), and then kernel2 creates data set B from data set A (consumes A and produces B).

In the preceding discussion, it did not matter what the data in the data sets were or what the kernels were doing to the data—it could have been rocket science, biology, or grocery lists. In the following example, it will be two matrices being multiplied—but think of combining grocery lists if that helps. Don't worry about understanding the details of the syntax of this example yet. It is given so that you can start comparing Kappa syntax with the discussion so that it starts to become familiar to you. This manual has plenty of pages describing the details of the syntax. Use the following pseudo code instead of you want:

```
    Start a CUDA context.
    Load the C and CUDA modules.
    Create variables A and B and initialize them.
    Create the variable C.
    Multiply A and B and get C using the matrixMul kernel from the CUDA module matrixMul.
```

The real Kappa process example is the following:

```
<kappa>
    // Start a CUDA context
    !Context ->  context;
    // Load the C and CUDA modules
    // (The file paths for {CMODULE} and {CUDAMODULE} must be setup
    // in the configuration files.)
    !C/Module -> testmodule={CMODULE};
    !CUDA/Module -> matrixMul = {CUDAMODULE};
    // Create variables A and B and initialize them
    // (The value for %sizeof{float} must be in the sizeof.conf configuration file)
    !Variable -> A(48,80,%sizeof{float});
```

```
        !C/Kernel MODULE='testmodule' -> randomInit (A,3840) [A];
        !Variable -> B(128,48,%sizeof{float});
        !C/Kernel MODULE='testmodule' -> randomInit (B,6144) [B];
        !Variable -> C(128,80,%sizeof{float});
        // Multiply A and B and get C
        !CUDA/Kernel GRID=[ 8, 5 ] BLOCKSHAPE=[ 16, 16 ]
                        SHAREDMEMORY=( 2 * 16 * 16 * %sizeof{float} )
          -> matrixMul@matrixMul(C,A,B,48,128) [ C = A B ];
</kappa>
```

This example shows that not only that C is dependent on A and B, but that the *matrixMul* kernel is dependent on the *matrixMul* module and that the *randomInit* kernel is dependent on the *testmodule* module. Kappa will schedule execution in the dependency order and will cancel execution of all dependencies if there is an execution failure such as an inability to load a module file. Note that there is an implicit dependency of modules, variables, and kernels on the context.

The developer must give the statements to Kappa in the correct dependency order so that Kappa can infer the correct dependency relationships when scheduling execution. If there is no dependency between statements, then Kappa will still schedule them in the order given but they may execute in some other order. For example, the allocation and initialization of the A and B variables could happen in any order or simultaneously as long as each variable's allocation precedes its initialization. Kappa Variable, Array, and Texture object types are not distinguished when it comes to resource dependency names. In other words, make that the names are different regardless of whether the object is a Variable, an Array, or a Texture.

Some other features are worth pointing out that made the above example possible: Kappa is allocating the requested host and/or device memory when required and is executing the appropriate CUDA data copy routines, as needed, to make sure that the correct data is in the correct host or device memory location prior to executing either C or CUDA kernels.

This is a good time to explain the general syntax of the Kappa process language given in the above and subsequent examples. See the Kappa Process Language section for complete details. Kappa Process language statements must be enclosed in paired tags:

```
        <kappa>
        ...Kappa language statements
        </kappa>
```

Almost all Kappa process statements start with an exclamation mark, '!', followed by a keyword, and the statement ends with a semicolon. There is one exception to Kappa process statements starting with an exclamation followed by a keyword and that is the decision statement, which starts with a question mark, '?', does not have a keyword but still ends with a semicolon. It is technically possible to put spaces between the exclamation and the keyword—Kappa will be forgiving and allow you to, but it is best form to have the exclamation immediately prior to the keyword so that the keywords are easy to see. For the Kappa process statements that start with an exclamation and a keyword, there may be attributes, followed by an arrow, '->'. If there are no attributes for a statement, then the arrow is sometimes optional. The remainder of a Kappa process statement takes different forms depending on what it is for. All of the allowed forms are listed in the Kappa Process Language syntax section.

The above example is incomplete for cases where subsequent processing would occur. While Kappa would automatically cleanup all allocated resources when it exits, it is a good practice to let these be freed as soon as possible. This means that statements to free the variables, unload the modules, and/or reset the context should also be given:

```
<kappa>
        !Free -> A;
        !Free -> B;
        !Free -> C;
        !CUDA/ModuleUnload -> matrixMul;
        !C/ModuleUnload -> testmodule;
        !ContextReset -> Context_reset;
</kappa>
```

It is also a good practice to let the Kappa background process know that no further commands will be queued for execution so that it can clean up and exit:

```
<kappa>
        !Stop;
        !Finish;
</kappa>
```

The Kappa process example just given shows that C kernels may be intermixed with CUDA kernels. Even though they are called C kernels and must be declared as 'extern “C”', the C kernel may be C++ kernels and, if the compiler supports it, the C kernels may implement OpenMP parallelism. Kappa supports a variety of ways of using C++ kernels that provide different benefits such as extending the keywords that are recognized or allowing for easier callbacks. Extending the keywords of Kappa is meant to allow extension of Kappa functionality but, of much more importance, to allow for creating keyword commands that support a particular subject field. The callback functionality is meant to allow for easy data access between the Kappa Process and the hosting program.

The synchronous C kernels execute on the background process thread that is usually tied to a CUDA context. This means that the synchronous C kernels have the ability to call all of the Kappa lower level API or CUDA driver functions, CUBLAS, CUFFT or other CUDA related libraries or functions such as OpenGL or Direct3D in addition to being able to call database or other functions and libraries.

A Kappa Process uses the program thread it is invoked on to parse, prepare for execution, and determine dependencies. It creates a separate, background process thread for scheduling and execution. This background process thread is the thread associated to a CUDA context—not the original program thread. Kappa synchronous C kernels, commands, Keyword commands, IOCallbacks, exceptions, and scheduling status handling are also running on this background process thread and have access to the CUDA context. These must all return in a reasonably timely fashion so that other scheduling, execution, exception handling, etc. may occur.

The previous process example is not very realistic in the sense that all of the sizes of the Variable objects, the grid's, block shapes, and shared memory are all static numbers. Kappa has full support for dynamic sizing calculations at process execution time. Kappa has full support for accessing as dynamic Value objects:

- the properties of the GPU,
- the properties of CUDA kernels as compiled on the particular GPU,
- system and user configuration,
- the integer or float content of Variable objects (even from GPU calculations),
- having C kernels, command::Keyword objects, or IOCallback objects set dynamic values such as data set sizes and dimensions,

and calculating and passing at execution time these parameters to

- Variable dimensions and sizes,
- kernel grid and block shape sizes, and shared memory allocations
- and as arguments to C and CUDA kernels.

It also supports evaluation of values and canceling execution of scheduled dependencies based on these dynamic values. For proper dependency schedule execution, these dynamic values must be stated as dependencies, in the same way that Variable objects are, in C and CUDA kernel calls.

The values for the dimensions of the A variable could be calculated as shown in the following example:

```
// Get BLOCK_SIZE value from configuration
!Value -> BLOCK_SIZE = %USER{BLOCK_SIZE};
```

This !Value statement creates a new value called BLOCK_SIZE and gets the numeric value for it from a configuration file that contains:

```
[/Kappa/Settings/USER]
CMODULE_PATH=commandtest/TestModule/.libs/
CMODULE={CMODULE_PATH}libTestModule.so
CUDAMODULE=matrixMul_kernel.cu
BLOCK_SIZE= %{BLOCK_SIZE}
[/Kappa/Translation/USER]
BLOCK_SIZE= 16
```

the “%USER{BLOCK_SIZE}” can be understood to be a translation (denoted by the '**%**' or percent sign) in the "/Kappa/Translation/**USER**" section of a configuration file that contains a label “**BLOCK_SIZE**” which in this case has the value 16. “%{BLOCK_SIZE}” and "%/Kappa/Translation/USER{BLOCK_SIZE}” also refer to the same thing since the default path is "/Kappa/Translation" and the default section is “USER”. (The “%sizeof{float}” that was used in the very first example can now be seen to be looking for a label called “float” in the "/Kappa/Translation/sizeof" section of a configuration file and substituting its content.)

The statement:

```
!Value -> WA =(3 * {BLOCK_SIZE}); /* Matrix A width  */
```

creates a value named WA which uses the {BLOCK_SIZE} USER setting “BLOCK_SIZE=%{BLOCK_SIZE}”. This shows that configuration values can refer to other configuration values—in this case the “%{BLOCK_SIZE}” configuration translation value that was previously discussed. (If configuration values refer to each other in a circular manner so that they could never be resolve, Kappa will currently give up trying after 128 attempts.) In this example, the WA value multiplies the BLOCK_SIZE by the number three to get a numeric value of 48.

The statement:

```
!Value -> HA = (5 * #BLOCK_SIZE); /* Matrix A height */
```

illustrates using the previous Value BLOCK_SIZE. Unless it is unambiguous that it is a Value being referred to and not a Variable or other name, then a Value must contain a pound sign, '#' to denote that it is a value. When defining a value, then the name given, such as "HA" in the current example, is obviously referring to a value and so the pound sign is unnecessary.

The configuration file shown above also defines the {CMODULE} and {CUDAMODULE} configuration values used in the very first process example.

To clarify the difference between a Value and a Variable, Variable objects in Kappa are blocks of memory that would usually hold multidimensional arrays of data but can contain anything that is consistent with their usage by the C and CUDA kernels that use them. Variable objects can be both input and output arguments to the C and CUDA kernels (a Variable can be a CUDA texture but the current version of CUDA only supports textures as inputs). They are automatically allocated and their data contents copied between host and device memory as needed. They can be host memory allocated with malloc, graphics resources (OpenGL or Direct3D) mapped by CUDA, or, more generally, are host or device memory allocated by CUDA and used as GPU memory, module variables, and textures.

Dynamic Value objects are arranged in hierarchical namespaces within a Kappa Process Namespace object and are only allowed as inputs to CUDA kernels while they can be inputs and outputs for C kernels. (Note, however, that Kappa supplies statements to create integer, float, or vector integer (Indices) Value objects from the contents of Variable objects and that command::Keyword or IOCallback functions have no constraints at all on what they can do with Value objects or Variable objects except those imposed by the current CUDA driver API.)

For further information about what a developer can do with Variable objects and Value objects, please see the command::Keyword keyword/CSV source code example.   It provides an illustration of how it is possible to extend the current capabilities of Kappa or provide subject area specialization. The keyword/CSV example provides a new keyword for reading csv data (in this example, with the CSV_DATA_TYPE attribute set to float and the file format set to CSV_TAB_DELIMITED) into a newly created variable, csv_variable, and returns the dimensions in the dynamic values, cols and rows:

```
!CSV CSV_DATA_TYPE=%KAPPA{FLOAT} CSV_TAB_DELIMITED=true ->
get(csv_variable,#cols,#rows,'test.csv');
```

The function name, "get", is not necessary in the example provided and could be omitted. Alternatively, it could be used by a different version of the CSV keyword to indicate whether to read or write the data from the file.

The CSV example illustrates that dynamics sizing is possible with Kappa. Also, please note that if the CSV command sets a canceled or failed status, then any subsequent scheduled commands that depend on its variable or values will have their execution canceled even if they are prepared and queued for execution in the process.

The final general topics worth mentioning in an overview is that:

- Kappa is designed to take full advantage of the Fermi GPUs by being able to schedule and run multiple concurrent CUDA kernels. For kernels that have independent data flow, just specify for them to be on different “streams” (even if they don't have independent data flow, it is OK to specify different streams—Kappa will sort it out). For CUDA kernels, if a stream is not specified, Kappa will automatically assign a stream from a pool of streams so that the kernels will execute concurrently.
- Kappa supports CUDA JIT compilation and loading of ".cu" or ".ptx" files and will prefer the ".ptx" file of the same age. Kappa can compile ".cu" files to ".ptx" files and load the resulting ".ptx file with JIT compilation. While Kappa can also load “FAT” or binary compiled files, NVIDIA suggests the use of ".ptx" files to automatically take advantage of new hardware capabilities. This functionality is available at all levels of the Kappa API.
- Kappa Context can use a previously created CUDA context (such as one created by the CUDA runtime). The statement: “!Context USE_CURRENT=true -> context;” creates the proper Kappa Context for the Process using an existing CUDA context, if it exists, otherwise it creates the necessary CUDA context.
- Kappa implements the CUDA driver version 3.0 API functionality including streams and asynchronous execution. It does not currently implement graphics resource mapping. (Graphic resource mapping may be added soon and the necessary Kappa API to support it is already present. Either user or Kappa supplied command::Keyword classes to implement OpenGL and Direct3D integration is the best route. Also, context pushing and popping, while implemented and available to the developer, is not currently used by the Kappa library.)
- Kappa provides access to the CUDA driver version 3.0 API functionality--all parameters may be specified even using the top level Kappa API except the graphics resource mapping mentioned previously.
- Kappa provides a flexible configuration system. It comes with a configuration file mechanism that allows for adding or updating configuration values in separate files. Extra configuration objects, such as one to retrieve from a relational database, may be easily hooked in. A configuration object to retrieve from relational databases may be provided with Kappa in the future. CUDA GPU properties are available as configuration values. All values defined in the NVIDIA cuda.h include file are provided in the cuda_translation.conf configuration file—please note the NVIDIA copyright notice in the cuda_translation.conf file.
- Kappa statements may be grouped into named subroutines that may specify a dynamic value of the number of times the subroutine statements should be executed. This loop value may be modified by the subroutine to shorten or lengthen the number of loops performed.
- Kappa statements may be grouped into ”functions” where the input and output arguments to the function are copied in and out (as appropriate) to and from the calling namespace and the function namespace(s). Each instance of a function call provides a separate function local namespace—in other words, the same function can be called repeatedly, at different levels of nesting, with no name collisions.
- Dependencies of statements are always tracked independently throughout a process, regardless of whether the statements are in subroutines or functions. This means that part of the statements in a subroutine or function may execute while other statements in that same

subroutine or function may be canceled because of what the statements depend on--even or especially dependencies external to the subroutine or function.

- Kappa subroutines and functions can be output to C++ shared library projects that can be compiled and distributed as shared libraries usable by Kappa. The only effect this has on the subroutines or functions is the speed with which they are loaded and prepared for use by a process (It is possible to tweak the values given to the attributes and arguments of the statements in the C++ output versions of the subroutines and functions so that the values are simplified to numeric constant values at C++ compile time instead of when being loaded or executed by a Kappa Process. This would cause these values to be static but also faster since they would be C++ constant values that would not be evaluated at runtime.)
- Kappa provides timers and the ability to check device memory usage to check for performance or memory problems. These are provided at all levels of the Kappa API's and have been used extensively when testing Kappa. (This, plus Kappa's constructor and destructor functionality, usually make cuda-memcheck superfluous.)
- Kappa may be used interactively or as a high level language for rapid application development (RAD) and for automation of testing.

Kappa Namespaces

All Kappa namespaces are in a hierarchical slash delimited path format. All namespaces beginning with "/kappa", in any combination of upper or lower case, may be used by the Kappa library and a developer should try understand the consequences of explicitly using such a namespace prior to its usage. Kappa Variable, Value, module, kernel, subroutine, function, and other names, either have an implicit full namespace qualified name or can be given an explicit namespace qualified name. The implicit name starts with the following root namespace path:

/kappa/context/main/

When the first !Context *context_name*; statement is given, it appends to this path:

/kappa/context/main/*context_name*

Any other !Context *other_context*; statements or !Function; call statements also append to the current namespace path while they are in effect—while the other context is valid or the function call is proceeding.

Any name that needs to be unique such as Kappa Variable, Value, module, kernel, subroutine, or function name, has the current namespace path prepended to it to construct a unique and fully qualified name. Kappa C/Kernel, CUDA/Kernel, and Print names have their arguments and map appended to create a unique command name for scheduling and tracking.

Kappa Configuration

The KappaConfigFile library may be used completely independently of the Kappa library. Alternatively, you may retrieve a copy of the KappaConfigFile object from the Kappa instance and use it for any purpose at the same time that Kappa is using it. KappConfig objects, such as KappaConfigFile, may be given additional KappaConfig objects that they will use when trying to find a value.

In addition to any directory paths passed to it, the KappaConfigFile object checks the “KAPPA_CONFIG_PATH” environment variable for a semicolon delimited list of directories to parse for configuration files. A configuration file must have a ".conf" file extension or it will be ignored.

Within a configuration file, empty lines or lines that have a pound sign (or a pound sign with spaces in front of it) at the beginning are ignored.

Configuration sections must be on a separate line and must have square brackets surrounding them. The section name could be any text string but supported values start with a slash and resemble a *nix file path—spaces are stripped off of the beginning and the end of the text string:

[/This/is/my/section]

Within a section, there should be labels followed by an equal sign, '=', followed by a value or labels followed by an ampersand equal sign, '&=', followed by a value. Any spaces at the beginning or the end of the labels are stripped off. Any spaces at the beginning or the end of the value are also stripped off. (If it is really necessary preserve spaces at the beginning and end of the values, use quotes around the values and then strip off the quotes.)

If an equal sign is used, then, if that section and label combination appear in multiple locations (i.e. in the same file or in different files or in different files in different directories), then the last one encountered is the value stored for that section and label. KappaConfigFile prints a warning to standard error whenever a section and label value are overwritten in this manner. There may be no way to determine which of the possible values will the result value so it is best to try to avoid this situation.

If an ampersand and equal sign combination is used, then the value is appended to the preceding value(s) with a newline character preceding it. This format is useful for putting the contents of a file into a single configuration value. Please look at the kappa_project.conf file for an example of how this may be used.

Suggested configuration directories

For Microsoft Windows, the recommended (and default) configuration file directories are:

"%CommonProgramFiles%\Kappa\Conf.d”

or

“%ALLUSERSPROFILE%\Kappa\Conf.d”

for system wide configuration files and

“%APPDATA%\Kappa\Conf.d"

for user specific files, where CommonProgramFiles, ALLUSERSPROFILE, and APPDATA are standard Microsoft Windows environment variables defining the standard Microsoft Windows path locations.

For all other platforms, the recommended (and default) configuration file directories are:

"/etc/kappa.d”

for system wide configuration files and

“~/.kappa.d"

for user specific files.

Kappa configuration files and section names

Kappa reserves any section name starting with "/kappa", with any combination of lower or upper case letters, for its use. The developer may change the values in these sections as appropriate.

Supplied Configuration File	Section Name	Configuration Section Description
cuda_translation.conf	/Kappa/Translation/CUDA	Provides translation of enumerations and macro definitions from cuda.h.
kappa_translation.conf	/Kappa/Translation/KAPPA	Provides translation of enumerations and macro definitions for the Kappa library.
sizeof.conf	/Kappa/Translation/sizeof	Provides byte sizes of some common data types.
	/Kappa/Translation/USER	Available for user defined translations.
kappa.conf	/Kappa	Used by the Kappa library Kappa class.
kappa.conf	/Kappa/CUDA/Module	Used by the Kappa library kappa::CUDA::Module class.
kappa.conf	/Kappa/C/Module	Used by the Kappa library kappa::C::Module class.
kappa_project.conf	/kappa/project	Used by the Kappa library kappa::Process class for creating project files for output routines.
	/Kappa/Settings/USER	Available for user defined setting values.
keyword.conf	/kappa/keywords	If enabled by a method of the kappa::Process class, defines keywords to load. This section should only consist of labels in the form: keyword&=CSV where CSV is a new keyword and kappa::Process will assume that there is a section named: [/kappa/keyword/CSV] or equivalent for the new keyword.
keyword.conf	/kappa/keyword/CSV	A section for automatically loading the shared library that provides the kappa::command::keywordCSV example class. NOTE: THE KEYWORD CLASS SHARED LIBRARIES MUST HAVE BEEN COMPILED WITH THE SAME ABI (APPLICATION BINARY INTERFACE) VERSION OF KAPPA. Consider carefully whether to use this feature in this manner. See the section on the kappa::command::keywordCSV example class for other, safer methods.

CUDA GPU Properties

The properties and attributes of the GPUs can be retrieved using the Kappa API access to the cudaGPU class. These properties are also made available to use for dynamic sizing as either:

/Kappa/CUDA/GPU/Current#<*Property*>

or

%/Kappa/CUDA/GPU/<*GPU#*>{<*Property*>}

where <*GPU#*> is the GPU associated with kappa::Process and <*Property*> is one of the following CUDA properties:

CUDA <*Property*>	**Description**
Name	Device name string
Major	Major revision number
Minor	Minor revision number
GlobalMemory	Total amount of global memory in bytes
ConstantMemory	Total amount of constant memory in bytes
SharedMemoryPerBlock	Total amount of shared memory per block in bytes
RegistersPerBlock	Total number of registers available per block
WarpSize	Warp size (currently always 32)
MaxThreadsPerBlock	Maximum number of threads per block
MaxThreads	Maximum sizes of each dimension of a block (vector integer Value of three numbers)
MaxGridSize	Maximum sizes of each dimension of a grid (vector integer Value of three numbers)
MemoryPitch	Maximum memory pitch in bytes
TextureAlignment	Texture alignment in bytes
ClockRate	Clock Rate (in hertz)
MultiProcessorCount	Multiprocessor Count
ConcurrentCopyExecute	Concurrent copy and execution (boolean)
KernelTimeout	Kernel timeout (boolean)
Integrated	Integrated (boolean)

CUDA <*Property*>	**Description**
CanMapHostMemory	Can Map Host Memory (boolean)
ComputeMode	Compute Mode (enumeration)
MaximumTexture1DWidth	Maximum texture one dimensional width
MaximumTexture2DWidth	Maximum texture two dimensional width
MaximumTexture2DHeight	Maximum texture two dimensional height
MaximumTexture3DWidth	Maximum texture three dimensional width
MaximumTexture3DHeight	Maximum texture three dimensional height
MaximumTexture3DDepth	Maximum texture three dimensional depth
MaximumArrayWidth	Maximum texture array width
MaximumArrayHeight	Maximum texture array height
MaximumArraySlices	Maximum texture array slices
SurfaceAlignment	Surface alignment in bytes
ConcurrentKernels	Whether concurrent kernel execution is supported (boolean)
ECCSupported	Whether ECC is supported (boolean)

Please refer to the CUDA Reference Manual or Programming Guide for further details on these properties and their uses.

CUDA/Kernel Attributes

The attributes of a kernel, as it is actually compiled and loaded on the GPU to execute, can be retrieved by executing the statement:

!CUDA/Kernel/Attributes MODULE=<*module_name*> -> <*kernel_name*>;

where <*module_name*> is the module name given and <*kernel_name*> is the name of the kernel. This create the following Value objects:

/kappa/CUDA/<*module_name*>/<*kernel_name*>#MaxThreadsPerBlock

/kappa/CUDA/<*module_name*>/<*kernel_name*>#StaticSharedMemory

/kappa/CUDA/<*module_name*>/<*kernel_name*>#RegistersPerThread

/kappa/CUDA/<*module_name*>/<*kernel_name*>#ConstantMemory

/kappa/CUDA/<*module_name*>/<*kernel_name*>#ThreadLocalMemory

/kappa/CUDA/<*module_name*>/<*kernel_name*>#PTXVersion

/kappa/CUDA/<*module_name*>/<*kernel_name*>#BinaryVersion

Please refer to the CUDA Reference Manual or Programming Guide for further details on these attributes and their uses.

Context Attributes

The attributes at a context level can be retrieved by executing the statement:

!Context/Attributes -> <*context_name*>;

where <*context_name*> is the name of the current context. This create the following Value objects:

/kappa/CUDA/<*context_name*>#MemoryFree

/kappa/CUDA/<*context_name*>#MemoryTotal

/kappa/CUDA/<*context_name*>#MemoryUsed

/kappa/CUDA/<*context_name*>#APIVersion

/kappa/CUDA/<*context_name*>#DriverVersion

/kappa/CUDA/<*context_name*>#CacheConfig

/kappa/CUDA/<*context_name*>#DeviceID

/kappa/CUDA/<*context_name*>#ThreadStackSize

/kappa/CUDA/<*context_name*>#PrintfFIFOSize

/kappa/CUDA/<*context_name*>#HeapSize

Please refer to the CUDA Reference Manual or C Programming Guide for the cuMemGetInfo, cuCtxGetApiVersion, cuDriverGetVersion, cuCtxGetCacheConfig, cuCtxGetDevice, and cuCtxGetLimit functions respectively for details for these values.

Kappa Core API

At its core level, Kappa provides objected-oriented encapsulation of the CUDA functionality. The purpose of this part of the Kappa library is to provide integrated encapsulation of the CUDA functionality. This means that it should setup reasonable defaults and fail-safe behavior so that fewer attributes need to be specified. It also means that most details of CUDA functionality should occur by default with minimal specification by the developer such as properly allocating and freeing memory, efficiently copying data, preparing kernels for launch and synchronizing those activities.

The core level of Kappa should never prevent access to CUDA functionality. It is not meant to hide the functionality of CUDA via encapsulation but rather to make the CUDA functionality integrate together in the manner that it was designed and documented it to do.

Actually, none of Kappa should prevent access to CUDA functionality. It is explicitly designed to be extensible with access to core Kappa and CUDA functionality. This is so that, if, by oversight, access is not available from some part of Kappa, an extension can be developed, by any adequate developer, that rectifies the shortcoming.

This following table provides a summary overview of the Kappa core functionality. Please refer to the Kappa Reference Manual for further details.

Kappa Core Classes	**Description**
Kappa	The main class for Kappa.
cudaGPU	Provides access to CUDA GPU properties.
ProcessControlBlock	Main encapsulation for background execution, scheduling, and commands. This module creates and tracks Context objects.
Context	Main interface for creating and retrieving core objects such as Variable, Array, Stream, Module (C and CUDA), and Kernel(C and CUDA) objects and for synchronization.
Variable	Provides encapsulation for host and device memory (which inherits functionality from LocalMemory, DeviceMemory, and DeviceTexture).
Array	Provides access to the CUDA array functionality.
Stream	Encapsulates a CUDA stream.
Timer	Provides timer functionality based on CUDA events.
Event	Provides the CUDA basis for timers and synchronization.
CUDA::Module	CUDA module compilation, loading, and access to Module variables, textures, and, kernels.
CUDA::Kernel	CUDA kernel call setup and launch.
C::Module	C module loading and access to kernels.
C::Kernel	C kernel call setup and launch.

Kappa Command Queue and ProcessControlBlock

The Kappa and ProcessControlBlock classes provide a background command queue with a separate execution thread for each GPU. The execution thread for each GPU has the association to a CUDA context and so each one of these can be considered a GPU/CPU process that runs CUDA and C kernels. Commands to schedule on these processes must inherit from either the kappa::Command or the kappa::command::Keyword classes. The scheduler for each process maintains a command input queue, a command paused queue, and a command running queue.

The kappa::Process class receives exception and status notifications from these schedulers and the command execution. The kappa::Process class can pass along these exception and status notifications. See the section on Errors and Testing or the Kappa Reference Manual for more information.

Kappa currently supports the following set of possible command statuses:

- READY
- RUNNING
- PAUSED
- CANCELED
- FINISHED
- FAILED

This core level of Kappa functionality does not provide any scheduling relationships between commands. The only scheduling provided at this level is the CUDA core functionality that constrains CUDA functions on the same stream to execute in order. To have commands works together to perform tasks in the right order, please use the Kappa Process class. It is unsupported by Psi Lambda to mix direct usage of the Kappa command queue with the use of the Kappa Process class.

An example program using the core Kappa functionality is documented in the Kappa Reference Manual. This example program also shows the usage of timers and tracking of Context memory usage.

Errors and Testing

Kappa supplies various mechanisms for performance tracking, error exception handling, and tracing to make it easier to localize errors or perform testing.

Print

The Kappa Process Print statement may be used to display Value objects and to track progress of Kappa Process execution.

Timers

The statement:

!Timer -> <*timer_name*>;

will, the first time it is given, create a timer. The subsequent times it it is given, it will output to standard error the elapsed time since the timer was last invoked:

Processing time: 42.984 (ms)

The times may include scheduling delays, system overhead and delays, etc.

Context memory usage

The statement:

!Context -> <*context_name*>;

will, if if is subsequently repeated in the same Kappa Process, report to standard error the device memory usage:

Device: Starting Free Memory: 514719744

```
        Ending:  Free Memory: 514719744
          Difference Memory: 0
                     Total: 536674304
                      Used: 146748
```

When the “!Context” statement with the same context name is given (repeated), it is given a “final” dependency status. This means that it is deferred for execution until all other dependencies on that named context have finished.

ExceptionHandler

The developer may instantiate a sub-class of the kappa::ExceptionHandler class and use the kappa::Process.SetExceptionHandler method to register to receive exception notifications from the Kappa Process.

SetQueueResults and GetResult

The developer may call the kappa::Process.SetQueueResults method to have kappa::Process start recording command status results to a queue and may call the kappa::Process:GetResult method to retrieve these command status results from a queue.

KAPPA_TRACE

Setting the KAPPA_TRACE operating system environment variable will have the Kappa Process report command statuses as they occur. These statuses report information similar to:

```
Preparing:    !C/Kernel -> CheckGold@(A,B,C,#HA,#WA,#WB,#HC,#WC) [= A B C, = #HA #WA #WB #HC #WC];

Queueing:     !C/Kernel -> CheckGold@(A,B,C,#HA,#WA,#WB,#HC,#WC) [= A B C, = #HA #WA #WB #HC #WC];

Finished      kappa::command::C::kernel /kappa/context/main/context/CheckGold@(A,B,C,#HA,#WA,#WB,#HC,#WC) [= A B C, = #HA #WA #WB #HC #WC] 113808

Preparing:    !Free -> A;

Queueing:     !Free -> A;

Canceled      kappa::command::VariableDone /kappa/context/main/context/A 113808
```

Please note that it is fine for commands to respond with a CANCELED instead of FINISHED status. Most commands that finalize (free) resources should respond with a CANCELED status so that further processing with those resources does not occur.

Please also note that full, internal, namespace name is usually what is reported in this trace. For example, the Variable "A" is reported as "/kappa/context/main/context/A" and the C/Kernel "CheckGold" is reported as "/kappa/context/main/context/CheckGold".

For a FINISHED or CANCELED command, the current CUDA device memory allocations is also reported. The 113808 number in the above example is the size, in bytes, of device memory that has ever been allocated at the time the command completed. This number is not the same as the amount device memory in actual use—it is the accumulation of all allocations—disregarding the freeing of the device memory.

Kappa Process Language

The Kappa Process processes may be defined using either text statements or C++ Instruction objects. Since a simple and sensible way to get started with or to generate the C++ Instruction objects is to create Kappa Process processes using text statements and then convert them to projects of C++ source code using Instruction objects, this section will mainly cover the text statements method of defining processes.

Kappa Process Language tag structure

Kappa provides a language for defining processes that can combine CPU and GPU operations in dynamic, coherent ways to implement tasks. Kappa Process language statements must be enclosed in paired tags:

<kappa>

...Kappa language statements

</kappa>

Kappa Process will ignore anything that is not enclosed in tags. Kappa will ignore any tags that it does not recognize. Tags can not currently be nested. The current tags that Kappa Process will currently utilize are either "<kappa>" tags. Other tags will be added by Kappa in the future.

Attributes may be included in the opening kappa tag. If Kappa does not recognize an attribute, it will be ignored. The attributes that Kappa Process currently recognizes and uses are used to define the block of statements within the kappa tags as either a Kappa Process subroutine or function. If no subroutine or function attribute is present, then the block of statements within the kappa tags is considered to be statements to be prepared for execution:

<kappa>

...Kappa language statements to immediately prepare and queue for execution

</kappa>

RememberAnonymous and DoNotExecute

The kappa::Process.DoNotExecute and kappa::ProcessRememberAnonymous methods may be used to tell kappa::Process not to execute these statements and/or to remember these as a subroutine named “main”.

Subroutine and Function Definition

If the subroutine attribute (and subroutine name) are given, then the statements are not prepared for execution but are instead stored for later invocation as a subroutine:

> <kappa subroutine=*subroutine_name*>
>
> *...Kappa language statements to store for later invocation as a subroutine*
>
> </kappa>

If the function attribute (and function name) are given, then the statements are not prepared for execution but are instead stored for later invocation as a function:

> <kappa function=*function_name* arguments="*arguments*" map="*map*">

OutputRoutines, OutputRoutine, and LoadRoutine

The kappa::Process.OutputRoutines, kappa::Process.OutputRoutine, and kappa::Process.LoadRoutine may be used to output these subroutines and functions to C++ files and to load them from a compiled shared library. The kappa::Process.OutputRoutines and kappa::Process.OutputRoutine methods will overwrite any existing files of the same names in the specified output directory. Please note that Psi Lambda LLC considers the source code produced by the kappa::Process.OutputRoutines and kappa::Process.OutputRoutine methods to be copies of the Kappa Process statement files which retain the copyright of the original statement files. See the Kappa Reference Manual for more details.

Kappa Process function attribute arguments and input/output maps

The arguments and map attributes are optional but are usually necessary. The arguments are the comma separated list of local argument identifiers. They provide a local name for a Variable or Value. If they are a dynamic local Value, then they must contain (or start with) a pound sign: “#local_value”. If they are a local Variable name, then it should only consist of alphanumeric characters and the underscore character.

The input/output ”map” attribute value is a comma delineated list. Each Variable or Value name given in a map should match the name given as a argument. Each item in the list has items on the left of an equal sign that are for output and items on the right of the equal sign that are for input. If no equal sign is present then the items are for both output and input. In the following example:

> A = B C, #area = #row #col, D

A, #area, and D are output items (Variable objects and Value objects) and B, C, #row, #col, and D are input items (Variable objects and Value objects). Yes, D was listed twice—it is both an input and output item (Variable in this case). A, B, C, D are Variable objects and #area, #row, and #col are Value objects.

Inputs for a function are copied from the corresponding external (calling) Variable or Value prior to function statements executing. Outputs for a function are copied to the corresponding external (calling) Variable or Value after function statements finish executing. So, to illustrate, the Variable B is copied from some (unknown at this point) Variable when the function is called and the Variable A is copied to some (unknown at this point) Variable prior to the function returning. The same for the remaining Variable objects and Value objects. For an input Value, the corresponding calling value could be a static string or numeric value—it does not have to be a dynamic Value.

(A local Variable or Value name could consist of a fully qualified namespace Variable or Value name with slash characters delineating the namespace path but that can not truly be considered to be a local Variable or Value. Consider carefully the effects if this is done—the calling Variable or Value may be copied to this “local” Variable or Value prior to the function statements executing and the “local” Variable or Value may be copied to a calling Variable or Value after the function statements have finished executing, depending on the contents of the input/output ”map”.)

Kappa Process Language comments

Kappa supports either 'C' style comments of the forms:

```
/* Comment
*/
```

or the newer single line comments:

```
// Comment
```

Kappa Process Instructions

This section may be skipped if a user or developer only wishes to define Kappa Process processes using statements.

All Kappa Process statements are parsed into Kappa Process Instruction objects. Kappa Process subroutines and functions are stored as std::vector objects of these Instruction objects. Kappa Process Instruction objects have a keyword, a name, Attributes, and Arguments. Kappa Process Instruction objects may also store the unparsed form of the arguments and map and the original, unqualified form of the name.

It is possible for a developer to directly create Kappa Process processes at the Instruction level. The simplest way to get started in this manner is to create a Kappa Process using statements and use the example program to output these to C++ source code. The kappa::Process.LoadRoutine and kappa::Process.ExecuteRoutine or kappa::Process.ExecuteFunction methods may then be used to load and execute these instructions as a subroutine or function.

An example of creating a routine (named “myroutine”) with an instruction is the following:

```
// Create the Attributes object and add the attributes to it.
Attributes *attribute_Context_context_1 = new Attributes();
(*attribute_Context_context_1)
.Add(string("CONTEXT_FLAGS"), (unsigned int)CU_CTX_SCHED_YIELD)
.Add(string("CUDA_CONTEXT_TYPE"), (unsigned int)CUDA_CONTEXT)
.Add(string("REPORT_MEMORY"), true);
```

```
    // Unparsed arguments and Unparsed map
    // These unparsed arguments and map may, for certain instructions,
    // be used to create a unique name for the resulting Command
    string upa_Context_context_1 = string("");
    string upm_Context_context_1 = string("");
    // Create the Arguments object and add the arguments to it.
    Arguments *argument_Context_context_1 = new Arguments();

    // Create the Instruction with the Attributes and Arguments
    Instruction *instruction_Context_context_1 =
    new Instruction(string("Context"), string("context"),
    attribute_Context_context_1,argument_Context_context_1,
    upa_Context_context_1,upm_Context_context_1);

    Subroutine *routine = new Subroutine(string ("myroutine"));
    (*routine).Add(instruction_Context_context_1);

    Attributes *attribute_Value_MyOtherNamespace_value_1 = new Attributes();

    string upa_Value_MyOtherNamespace_value_1 = string("42");
    string upm_Value_MyOtherNamespace_value_1 = string("");
    Arguments *argument_Value_MyOtherNamespace_value_1 = new Arguments();
    (*argument_Value_MyOtherNamespace_value_1)
    .In((unsigned int)42);
    Instruction *instruction_Value_MyOtherNamespace_value_1 =
    new Instruction(string("Value"), string("/My/Other/Namespace#value"),
    attribute_Value_MyOtherNamespace_value_1,argument_Value_MyOtherNamespace_value_
1,
    upa_Value_MyOtherNamespace_value_1,upm_Value_MyOtherNamespace_value_1);

    (*routine).Add(instruction_Value_MyOtherNamespace_value_1);
```

```
        (*routines)[string ("myroutine")] = routine;
```

This illustrates creating a subroutine with an instruction to create a context and a Value. The equivalent statement (indeed—it is the source statement for the above C++) is:

```
<kappa>
!Context CUDA_CONTEXT_TYPE=%KAPPA{CUDA_CONTEXT}
    CONTEXT_FLAGS=%CUDA{CU_CTX_SCHED_YIELD}
    REPORT_MEMORY=true
 ->  context;
!Value -> /My/Other/Namespace#value = 42;
</kappa>
```

Kappa Process Language syntax

For any form of Kappa statement, anywhere there is a space, any other white space can occur such as a new line. Kappa statements currently have only two basic forms. The first form is for declarative statements and the following shows all of the valid declarative statement forms:

```
!Keyword attr="value" ... -> name@module(arguments) [map];

!Keyword -> name@module(arguments) [map];

!Keyword attr="value" ... -> name@module(arguments);

!Keyword attr="value" ... -> name@module = from;

!Keyword -> name@module = from;

!Keyword -> name@module(arguments);

!Keyword attr="value" ... -> name@module;

!Keyword attr="value" ... -> name(arguments) [map];

!Keyword -> name(arguments) [map];

!Keyword attr="value" ... -> name(arguments) = from;

!Keyword attr="value" ... -> name(arguments);

!Keyword attr="value" ... -> name = from;

!Keyword -> name(arguments) = from;

!Keyword -> name(arguments);

!Keyword -> name = from;

!Keyword attr="value" ... -> name;

!Keyword -> name;

!Keyword name(arguments);

!Keyword;
```

where 'attr=”value” ...' denotes that there are zero or more attribute/value pairs. The quotes around the attribute value are usually optional and sometimes undesired. The arguments and map are of the same format as discussed previously. However, the arguments and maps given in these declarative statements must have the parenthesis or brackets while parenthesis and brackets are not used for tag attributes.

In the declarative statements, the keyword is always mandatory. So the minimal declarative statement syntax possible is:

    !Keyword;

To the Kappa parser this is all that is necessary but it parses the remaining elements, if they are present in the correct forms as shown above. If elements are present, but not in the correct form(s), then the Kappa parser will complain. Kappa keywords enforce their own syntax and complain if elements they require are missing.

The various forms shown above is for the convenience of the user—the number of different possibilities is actually less than shown. The “module” element show above, can be given as the attribute “MODULE” instead. The ”from” element constitutes the last of the (comma delimited) “arguments” element.

The second basic form is for decision statements:

    ? logical_expression -> arguments;

The arguments for a decision statement are a list of Variable objects and Value objects to make dependent on the decision. The logical_expression must only contain numbers, configuration values, Value objects, and logical or numeric operators. All configuration values and Value objects must be resolvable to numeric or boolean values. The logical_expression is evaluated and, if it is zero, the decision statement reports a CANCELED completion status. This cancels execution of all statements that are dependent on the Variable objects or Value objects in the arguments list. It reports a FINISHED completion status if the logical_ expression evaluates to nonzero. It reports a FAILED completion status (which will have the same effect as a CANCELED completion status) if the logical_expression can not be evaluated to an integer or boolean value (for example if configuration values or Value objects can not resolve or that resolve to text values that are not convertible to numeric values).

For boolean values for attributes or Value objects, “true” or “on” and “false” and “off” or numeric values may be given. For Indices Value objects, a comma delimited list of integer values with optional spaces may be given. For logical and numeric expressions, configuration values and Value object may be used, as long as they are resolvable. For logical and numeric expressions, parenthesis are supported and outer, enclosing parenthesis may be mandatory for proper classification.

An 'if' function is supported in expressions. It is currently the sole mechanism, within a Kappa process, to conditionally set a Value to different values. It may be used in a !Value statement of the following form:

    !Value -> *value_name* = if(*condition* , *value_if_true* , *value_if_false*);

For logical and numeric expressions, the following binary operators are currently supported:

+ - * / ^ or and xor < > == <= >= !=

The following trigonometric functions are available: sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh. Also the following functions are available: log2, log10, log, ln, exp, sqrt, sign, rint, abs, min, max, avg, sum—where log is the same as log10, exp is *e* raised to the power of the argument, sqrt is the square root of the argument, sign returns -1 if the number is negative and +1 if the number is positive, and rint which rounds the argument to the nearest integer.

Built-in keywords

Kappa comes with a set of built-in keyword statements which are documented in the following sections. If a kappa::command::Keyword uses the same keyword as a built-in keyword statement, then it replaces the built-in keyword statement and correspondingly changes the Kappa Process functionality. Any such changes must be documented by the developers of the kappa::command::Keyword classes and can not be within the scope of any Psi Lambda Kappa documentation.

There are four main types of built-in keyword statements. They consist of resource creation, resource destruction, processing, and informational.

Resource Creation, Loading, or Access Keywords

Resource initialization statements create, load, or access a resource.

Context

Possible syntax:

!Context *attributes* -> *context_name*;

!Context *context_name*;

Possible attributes are:

USE_CURRENT

This may be either true or false (the default). This sets whether the Context statement should try to use an existing CUDA context or whether to always create a new one. If this is set to true, and an existing CUDA context is not available, then a CUDA context is created. Example:

USE_CURRENT=true

CUDA_CONTEXT_TYPE

This may be:

%KAPPA{CUDA_CONTEXT} which is an integer 0 (the default),

%KAPPA{OPENGL_CONTEXT},

%KAPPA{DIRECT3D9_CONTEXT},

%KAPPA{DIRECT3D10_CONTEXT},

%KAPPA{DIRECT3D11_CONTEXT}

This sets the type of CUDA context created. For graphics resource mapping and interoperability, the correct CUDA context type must be selected. Example:

CUDA_CONTEXT_TYPE=%KAPPA{OPENGL_CONTEXT}

CONTEXT_FLAGS

This is the CUDA context flags. You may use the cuda_translation.conf values in the form: (%CUDA{CU_CTX_SCHED} | %CUDA{CU_CTX_MAP_HOST}) to construct the flag value. The default value is shown in this example:

CONTEXT_FLAGS=(%CUDA{CU_CTX_SCHED} | %CUDA{CU_CTX_MAP_HOST})

REPORT_MEMORY

This may be either true (the default) or false. When set to true, the context tracks device memory allocation requests and displays them to standard error if the "!Context *context_name*;" statement is subsequently given. See the Errors and Testing section for further information.

CACHE

Sets the preferred cache type for the CUDA context. Set this to CU_FUNC_CACHE_PREFER_SHARED to prefer more shared memory and set this to CU_FUNC_CACHE_PREFER_L1 to prefer more L1 cache.

CACHE=%CUDA{CU_FUNC_CACHE_PREFER_SHARED}

Please see the description of the cuCtxSetCacheConfig function in the CUDA Reference Manual for a description of setting the cache preference.

STACK_SIZE

Sets the stack size for each GPU thread.

Please see the description of the cuCtxSetLimit function in the CUDA Reference Manual for a description of setting the stack size.

HEAP_SIZE

Sets the size of the heap for the device malloc() and free() calls.

Please see the description of the cuCtxSetLimit function in the CUDA Reference Manual for a description of setting the heap size.

PRINTF_FIFO_SIZE

Sets the size of the FIFO buffer for the printf() device calls.

Please see the description of the cuCtxSetLimit function in the CUDA Reference

Manual for a description of setting the FIFO buffer size.

Value

Create (or replace) a dynamic Value.

The possible syntax is:

!Value *attributes* -> *value_name*;

!Value *value_name*;

!Value *attributes* -> *value_name* = *from*;

!Value *value_name*= *from*;

where the *value_name* may be a Value name as discussed in the Overview or Namespace section and there are not currently any supported attributes. For extraction of values from a Variable, the possible syntax is:

!Value *attributes* -> *value_name*(slice_arguments) = *variable_name*;

!Value *attributes* -> *value_name*(slice_arguments, *variable_name*);

!Value -> *value_name*(slice_arguments) = *variable_name*;

!Value -> value_name(slice_arguments, variable_name);

The *value_name* may still be a Value name as discussed in the Overview or Namespace section. The *variable_name* is the name of an existing Variable that contains the values to extract.

For the first syntax form, where the values are not being extracted from a Variable, the *from* expression may be:

- configuration value(s),
- a numeric or string value,
- an Indices which is given in the form: [number, ...]. For example: [1,2,3,4] creates a vector value of four unsigned integer values (an Indices of length four), that has, in order, the values 1, 2, 3, and 4.
- an expression that is or evaluates to a numeric or string value (this can include a ternary operator),
- or another Value name.

For the form where values are being extracted from a Variable, the possible attribute that is currently supported is:

FROM_FLOAT

This may either be false (the default) which means that the values to be extracted are integer values or may be true so that the values are extracted from the Variable are “float” values. If more than one value is extracted, then the resulting data type is Indices which is a vector of unsigned integer values and float values are rounded to unsigned integer. If one value is extracted and the FROM_FLOAT attribute is true, then the resulting data type is the float representation of the Variable value. Example:

FROM_FLOAT=true

Retrieving Value objects from a Variable without the correct FROM_FLOAT attribute or if the elements of the Variable are not “simple” multidimensional arrays of integer or float values will lead to incorrect Value objects, may lead to program crashes, and is not supported by Psi Lambda. NOTE: if the Variable contains complex data structures, data structures that are padded or offset or contain values that are not integer or float in sequential format or if the FROM_FLOAT attribute is not given when the Variable contains float data then this will probably result in Value(s) that are incorrect and might result in program memory access violations (segmentation faults or “bus” errors). The feature of retrieving Value objects and Value Indices from Variable objects (along with the decision statement) is meant to allow the calculations of CUDA kernels to have (indirect) control of the sizing and execution of Kappa Process statements. However, as with any powerful feature, it possible for it to be misused.

The “slice_arguments” for extraction from a Variable depends on how many dimensions the Variable has:

- If the Variable is linear memory with a single dimension, then no arguments are necessary and the whole Variable objects contents are extracted into a Value Indices of the same linear length.
- If the Variable is linear memory with a single dimension, then a single argument can be given to specify the number of elements to extract from the Variable objects contents, starting with the first element. If an number larger than the length of the Variable objects length is given, then only the true number of elements are extracted.
- If the Variable is linear memory with a single dimension, then two arguments can be given to specify the starting (zero based index) element and the number of elements to extract from the Variable objects contents. If an number larger than the remaining length of the Variable objects length is given, then only the true number of possible elements are extracted.
- If the Variable is a multidimensional (two dimensions or more) array of memory, then an argument is needed for each dimension greater than one to specify the (zero based) index into the array for that dimension. Any number (greater than one) of dimensions is supported. So for a Variable with N dimensions, N-1 arguments are required to be specified. These arguments are given in order of higher dimension (i.e., y, z, etc.). The supported number of arguments given for extracting from a Variable with N dimensions is:
    - specifying N-1 arguments extracts the complete first dimension (row) specified by the N-1 index location,
    - specifying 1 + N-1 arguments specifies the number of elements to extract from the row and then the N-1 index location. This extracts the number of elements specified in the first dimension (row) specified by the N-1 index location,
    - specifying 2 + N-1 arguments specifies the starting (zero based index) element and the number of elements to extract from the row and then the N-1 index location. This extracts the number of elements specified in the first dimension (row) starting at a row element at the specified N-1 index location.

The slice argument may be a Value Indices of the proper length.

Examples for a linear memory (one-dimensional) Variable:

```
// Extract the full Variable as integer an (Indices) Value object
!Value -> ivalues() = MyVariableIndices;
// Extract three elements from the Variable as an integer (Indices) Value
!Value -> ivalues(3) = MyVariableIndices;
// Assuming at least two float elements, extract the second element from the Variable
// as a float  Value
!Value FROM_FLOAT=true-> fvalues(1,1) = MyVariableFloat;
```

Examples for an array of integer memory (three-dimensional) Variable:

```
// Extract a full row from the Variable as an integer (Indices) Value
// This specifies the complete x row at the fourth (3) y index and the fifth (4) z index.
!Value -> ivalues(3,4) = MyVariable3DIndices;
// Extract part of a row from the Variable as an integer (Indices) Value
// This specifies 2 elements of the x row at the the fourth (3) y index
// and the fifth (4) z index.
!Value -> ivalues(2,3,4) = MyVariable3DIndices;
// Extract two elements of a row from the Variable as an integer (Indices) Value
// This specifies the second and third elements (1 and 2) of the x row
// at the the fourth (3) y index and the fifth (4) z index.
!Value -> ivalueIndices = [ 1, 2, 3, 4 ];
!Value -> ivalues(#ivalueIndices) = MyVariable3DIndices;
```

This last example illustrates how a Value Indices may be given as an argument anytime the Value Indices contains the proper number of and value of elements to correspond to the correct argument values. Since these Value Indices could have, in turn, been extracted from other Variable objects, this allows for true dynamic control of the Kappa Process—especially since Variable declaration arguments and CUDA/Kernel GRID and BLOCKSHAPE launch attributes may also be Value Indices.

Variable

Create a Variable. A Variable can be a “blob” of data—with no dimensional sizes given to Kappa (technically a ”blob” is a one-dimensional Variable with one element whose size is the amount of memory allocated and copied by the Variable—this means that Kappa is totally unaware of any internal structures) or a Variable can be declared as containing certain dimensions of certain sizes. Kappa supports Variable objects of any dimension greater than or equal to one and any dimension may be of any size supported by an unsigned integer value.

Kappa Variable objects may be assigned streams which are used to allow asynchronous copying.

To the Kappa Process, except for when Value objects are extracted from a Variable, Variable objects are treated as opaque. A Variable objects internal contents are not “visible” to Kappa Process statements—only to Kappa core library methods C and CUDA kernels, user IOCallbacks and kappa::command::Keyword objects, etc.

Possible syntax:

> !Variable *attributes* -> *variable_name*(*arguments*);
>
> !Variable *variable_name*(*arguments*);
>
> !Variable *attributes* -> *variable_name*;
>
> !Variable *variable_name*;

Possible attributes are:

VARIABLE_TYPE

> This is the type of Variable to create and provide allocation and copying behavior for. The default is Local or zero. Possible values are:
>
> Local=0
>
> > Provide for initial allocation using host memory but use CUDA memory allocation functions. This is “pinned” or “non-paged” memory. This type of memory is a scarcer resource than LocalOnly memory.
>
> LocalAndDevice=1
>
> > Provide for initial allocation using integrated or mapped memory and use CUDA memory allocation functions. This will fallback to separate host and device allocation if the GPU does not support integrated or mapped memory. This type of memory is a scarcer resource than LocalOnly memory.
>
> LocalOnly=2
>
> > Provide for initial allocation using host memory and use standard malloc memory allocation function. This is “non-pinned” or “paged” memory. This type of Variable will not copy to or from a GPU as fast as other VARIABLE_TYPE types.

LocalToDevice=4

Provide for initial allocation using host memory but use CUDA memory allocation functions with the WriteCombined flags set. This is”pinned” or “non-paged” memory. This type of memory is a scarcer resource than LocalOnly memory but is optimal for staging data from the host to the GPU.

Device=5

Provide for initial allocation using device memory and use CUDA memory allocation functions. If needed, host memory will be allocated as Local--see above.

DevicePitch=6

Provide for initial allocation using device memory and use CUDA memory allocation functions with pitch allocation: cuMemAllocPitch. If needed, host memory will be allocated as Local--see above.

DeviceOnly=7

Provide for initial allocation using device memory and use CUDA memory allocation functions. If needed, host memory will be allocated as LocalOnly--see above.

Example:

```
VARIABLE_TYPE=%KAPPA{Local}
```

COLUMNS

The number of (virtual) columns in the Variable. For a Variable that will hold a dataset with rows and columns and the columns are of different widths, this allows specifying the number of columns contained in the Variable. Example:

```
COLUMNS = 4
```

NULL_BLOCK_WIDTH

Used to change the block width used to store the NULL value mask for the Variable from 8 bit words to 32 bit words. Example:

```
NULL_BLOCK_WIDTH = 32
```

STREAM

The name of a stream to associate with the Variable for doing asynchronous copies (and asynchronous or concurrent kernel launches if the same stream name is also given to a CUDA/Kernel). The Kappa Stream object and CUDA stream and any necessary CUDA events are automatically created and managed. Example:

```
STREAM=mystream
```

DEVICEMEMSET

A boolean value specifying whether to perform a cuMemsetD* to clear the device memory. Default value is false. This assumes that it is desired to initialize the memory to zero and that either the dimensions and element size have been given or that the memory may be initialized to zero as 32 bit integer values. Example:

```
DEVICEMEMSET=true
```

The *arguments* consist of either the total size of the Variable (assumes a “blob” Variable of one dimension and element size of 1) or of the size of each dimension of the Variable followed by the element size (in bytes) of the Variable. Any number of dimensions greater than or equal to one is supported.

Array

Create an Array for use as a source for the Texture keyword statement, the Surface keyword statement, or with the CUDA cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAto*X*, cuMemcpy*X*toA, cuMemcpyAto*X*Async, or cuMemcpy*X*toAAsync routines (where *X* is one of H (host), D (device), or A (array)).

Possible syntax:

> !Array *attributes* -> *array_name* = *source_name*;
>
> !Array array*_name* = *source_name*;
>
> !Array *attributes* -> *array_name*;
>
> !Array array*_name*;

Source name may be the name of a Variable that contains data used to initialize the array.

Possible attributes are:

FLAGS

> The default value is zero. This may be set to one of the following:
>
> CUDA_ARRAY3D_2DARRAY or CUDA_ARRAY3D_SURFACE_LDST.
>
> This must be set to CUDA_ARRAY3D_SURFACE_LDST if the array will be used with a surface.

WIDTH

> This is an unsigned integer specifying the width of the array and has a default width of 1. Example:
>
> > WIDTH=2

HEIGHT

> This is an unsigned integer specifying the height of the array and has a default height of 1. If set to zero, then the Array is one-dimensional. Any nonzero value makes the Array to be at least two-dimensional—setting DEPTH to nonzero makes the Array three-dimensional and implies that height is nonzero. Example:
>
> > HEIGHT=2

ARRAY_FORMAT

> This is the data format of the array. You may use the cuda_translation.conf values in the form: %CUDA{CU_AD_FORMAT_FLOAT} (the default value). Possible values are:
>
> > CU_AD_FORMAT_UNSIGNED_INT8=1
> >
> > CU_AD_FORMAT_UNSIGNED_INT16=2
> >
> > CU_AD_FORMAT_UNSIGNED_INT32=3

CU_AD_FORMAT_SIGNED_INT8=8

CU_AD_FORMAT_SIGNED_INT16=9

CU_AD_FORMAT_SIGNED_INT32=10

CU_AD_FORMAT_HALF=16

CU_AD_FORMAT_FLOAT=32

Example:

ARRAY_FORMAT=%CUDA{CU_AD_FORMAT_FLOAT}

NUMBER_CHANNELS

This is an unsigned integer specifying the number of packed components (vectors, such as float2 or float4) of the array and has a default value of 1. This may also be set to 2 or 4.

DEPTH

This is an unsigned integer specifying the depth of the array and has a default depth of 0. If set to zero, then the Array is one-dimensional (if height is zero) or two-dimensional (if height is nonzero). Any nonzero value makes the Array to be three-dimensional and implies that height is nonzero. Example:

DEPTH=1

STREAM

The name of a stream to associate with the Array for doing asynchronous copies (and asynchronous or concurrent kernel launches if the same stream name is also given to a CUDA/Kernel). The Kappa Stream object and CUDA stream and any necessary CUDA events are automatically created and managed. Example:

STREAM=mystream

Texture

Create a Texture for use by a CUDA/Kernel. The name of the texture must match the name declared in the CUDA module.

Possible syntax is:

!Texture MODULE=*module_name attributes* -> *texture_name* = *source_name*;

!Texture *attributes* -> *texture_name@module_name* = *source_name*;

!Texture *texture_name@module_name* = *source_name*;

!Texture MODULE=*module_name attributes* -> *texture_name*;

!Texture *attributes* -> *texture_name@module_name*;

!Texture *texture_name@module_name*;

Source name may be the name of a Variable or Array that contains data used to initialize the Texture, as appropriate. A Variable may be used to initialize a linear, one-dimensional Texture. An Array is necessary to initialize a two-dimensional or three-dimensional Texture. The *module_name* may either be given as the MODULE attribute or may be appended to the *texture_name* with an '@' delimiter.

Possible attributes, besides the MODULE attribute, are:

AddressMode0

Sets the addressing mode for the first (zero) dimension of the texture reference. Possible values are: CU_TR_ADDRESS_MODE_WRAP (the default), CU_TR_ADDRESS_MODE_CLAMP, or CU_TR_ADDRESS_MODE_MIRROR. Example:

AddressMode0=%CUDA{CU_TR_ADDRESS_MODE_WRAP}

AddressMode1

Sets the addressing mode for the second (one) dimension of the texture reference. Possible values are: CU_TR_ADDRESS_MODE_WRAP (the default), CU_TR_ADDRESS_MODE_CLAMP, or CU_TR_ADDRESS_MODE_MIRROR. Example:

AddressMode1=%CUDA{CU_TR_ADDRESS_MODE_WRAP}

FilterMode

Sets the filtering mode used when reading memory via the texture reference. Possible values are: CU_TR_FILTER_MODE_POINT and CU_TR_FILTER_MODE_LINEAR (the default). Example:

FilterMode=%CUDA{CU_TR_FILTER_MODE_LINEAR}

TextureFlags

Sets flags for how the data is returned via the texture reference. Possible flags are: CU_TRSF_READ_AS_INTEGER and CU_TRSF_NORMALIZED_COORDINATES (the default). Example:

TextureFlags=%CUDA{CU_TRSF_NORMALIZED_COORDINATES}

Note: do not forget to add the name of the texture to the map of any dependent CUDA/Kernel statements.

Surface

Create a Surface for use by a CUDA/Kernel. The name of the surface must match the name declared in the CUDA module. The CUDA module must be compile for compute architecture 2.0 or higher (add *-arch=compute_20* to the *NVCC_PTX* entry in the *kappa.conf* configuration file).

Possible syntax is:

!Surface MODULE=*module_name* -> *texture_name* = *source_name*;

!Surface -> *texture_name@module_name* = *source_name*;

!Surface *texture_name@module_name* = *source_name*;

Source name must be the name of an Array to bind to the surface. The *module_name* may either be given as the MODULE attribute or may be appended to the *texture_name* with an '@' delimiter.

Note: do not forget to add the name of the surface to the map of any dependent CUDA/Kernel statements.

C/Module

Load a C or C++ module (shared library file or DLL).

Possible syntax is:

!C/Module *attributes* -> *module_name* = *file_name*;

!C/Module *attributes* -> *module_name(file_name)*;

The file name must be the name of the shared library or DLL. In the kappa.conf file, in the "/Kappa/C/Module" section, the OBJ_PATHS labels may be used to set paths to search for C/Module file names. A %s in the value for the OBJ_PATHS is replaced by the path to the executable file, if it was given as an argument to Kappa. If the OBJ_PATHS mechanism is not used, then an absolute file path name must be given.

The SHARED_OBJECT_EXTENSION label sets the extension for the module file. For Microsoft Windows, the extension should be ".dll", for Macintosh OS/X, the extension should either be ".dylib" or ".so", for most other operating systems the extension should be ".so".

There are no current attributes for the C/Module statement at this time.

CUDA/Module

Load a CUDA module.

Possible syntax is:

!CUDA/Module *attributes* -> *module_name* = *file_name*;

!CUDA/Module *attributes* -> *module_name(file_name)*;

The file name must be the name of a CUDA module file. In the kappa.conf file, in the "/Kappa/CUDA/Module" section, the OBJ_PATHS labels may be used to set paths to search for CUDA/Module file names. A %s in the value for the OBJ_PATHS is replaced by the path to the executable file, if it was given as an argument to Kappa. If the OBJ_PATHS mechanism is not used, then an absolute file path name must be given.

The file name may be a ".cu" source file, a ".ptx" intermediate file, or a CUDA binary object file. If a ".ptx" file of the same age or newer as a ".cu" file is found, then the ".ptx" file is preferred.

For a ".cu" source file, proper values for the CUDA_PATH and NVCC_PTX labels must be given in the "/Kappa/CUDA/Module" section of the kappa.conf file. The typical value for CUDA_PATH is "/usr/local/cuda". Typical values for NVCC_PTX are either:

```
NVCC_PTX=/usr/local/cuda/bin/nvcc -m32 -I. -I/usr/local/cuda/include -O3 -o %s.ptx -ptx
```

%s.cu

or

NVCC_PTX=/usr/local/cuda/bin/nvcc -m64 -I. -I/usr/local/cuda/include -O3 -o %s.ptx -ptx %s.cu

depending on whether the CUDA modules are compiled for 32 or 64 bit GPU address space. If the host machine is a 64 bit computer running a 64 bit operating system, then either value may be used.

(Kappa only supports the 32 bit GPU address space at this time.)

Also in the "/Kappa/CUDA/Module" section of the kappa.conf file are the default settings for JIT compilation. Settings these values in the kappa.conf file may make it unnecessary to set the attributes in the CUDA/Module statement. These labels and their default values are:

```
JITMaxRegisters=0
JITThreadsPerBlock=0
JITRecordCompileTime=false
JITOptimizationLevel=4
JITTargetFromContext=true
# The following %CUDA{} values come from [/Kappa/Translation/CUDA]
JITTarget=%CUDA{CU_TARGET_COMPUTE_13}
JITFallback=%CUDA{CU_PREFER_PTX}
```

Possible attributes are:

MODULE_TYPE

This sets the preferred module file type. Possible values are: CU_MODULE (the default), PTX_MODULE (the fallback if a ".cu" file is not found), ANY_MODULE, and FAT_MODULE. Example:

```
MODULE_TYPE=%KAPPA{CU_MODULE}
```

JITMaxRegisters

Please see the description of the cuModuleLoadDataEx function in the CUDA Reference Manual for a description of this attribute.

JITThreadsPerBlock

Please see the description of the cuModuleLoadDataEx function in the CUDA Reference Manual for a description of this attribute.

JITRecordCompileTime

Please see the description of the cuModuleLoadDataEx function in the CUDA Reference Manual for a description of this attribute which is named CU_JIT_WALL_TIME in that manual.

JITOptimizationLevel

Please see the description of the cuModuleLoadDataEx function in the CUDA Reference Manual for a description of this attribute.

JITTargetFromContext

Please see the description of the cuModuleLoadDataEx function in the CUDA Reference Manual for a description of this attribute which is named CU_JIT_TARGET_FROM_CUCONTEXT in that manual.

JITTarget

Please see the description of the cuModuleLoadDataEx function in the CUDA Reference Manual for a description of this attribute.

JITFallback

Please see the description of the cuModuleLoadDataEx function in the CUDA Reference Manual for a description of this attribute.

RECOMPILE

Specify true to force recompiling the ptx file even if it is newer. Example:

RECOMPILE=true

NVCC_OPTIONS

Set the string containing any additional nvcc options. The string will be stripped of anything that is not a dash, slash, underscore, alphanumeric, white space, equal sign ,dot, single quotes, or double quotes. Example:

NVCC_OPTIONS='-arch=compute_20'

CUDA/Variable/Module

Load a CUDA module variable.

Possible syntax is:

!CUDA/Variable/Module *attributes* -> *variable_name@module_name* = *from_variable*;

!CUDA/Variable/Module MODULE=*module_name attributes* -> *variable_name* = *from_variable*;

The *variable_name* must match the variable name declared in the CUDA module. The *from_variable* must be the name of a previously declared Kappa Variable.

There are no current attributes other than the MODULE attribute for the CUDA/Variable/Module statement at this time.

Note: do not forget to add the name of the variableto the map of any dependent CUDA/Kernel statements.

Keyword

Load module variable.

Possible syntax is:

> !Keyword *attributes* -> *new_keyword(shared_library)* = *factory_function*;
>
> !Keyword -> *new_keyword(shared_library)* = *factory_function*;
>
> !Keyword *attributes* -> *new_keyword(shared_library, factory_function)*;
>
> !Keyword -> *new_keyword(shared_library, factory_function)*;

By default, this statement is not enabled. See below for notes on why to not enable it and how to enable it anyway.

The *new_keyword* is registered as a new Kappa statement keyword. The *shared_library* may be an empty string, '', to try to located the *factory_function* in the main executable or its already loaded libraries. If the *shared_library* is not an empty string then it must be the complete file path name to the shared library containing the *factory_function* (and presumably to the kappa::command::Keyword class as well) for instantiating a kappa::command::Keyword subclass that implements the new keyword. The *factory_function* must be declared 'extern "C"' and return an instance of a subclass of the kappa::command::Keyword class cast to be type (void *).

Please see the keyword/CSV project source code files for implementation details. NOTE: using an external shared library that is dynamically loaded at runtime may cause the program to crash if the external shared library has not been compiled with the same ABI (application binary interface) version of the Kappa library. The Kappa library is ABI compatible throughout every major version number.. This means, for example, that all versions of the Kappa library 1.0 through 1.4 are ABI compatible. Using a shared library or DLL that is linked at program start up and setting the *shared_library* to an empty string or using the kappa::Process.RegisterKeywordCommand method provides safer functionality. Also, the kappa::Process.RegisterKeywordCommand method provides for passing a handle to user data or a pointer to a user object and is therefore the preferred method for adding Kappa Process keywords. (Psi Lambda LLC will not insist that you do not use this functionality but does highly discourage its use in uncontrolled or possibly insecure environments.) To enable this functionality, the kappa::Process.AddKeywordKeyword method must be called prior to use. Note that the kappa::Process.ProcessKeywordConfig method does not enable the Keyword statement (even though loading keywords from a configuration file suffers from the same problems). This a definite example of a situation where Psi Lambda LLC provides functionality but defaults to fail-safe settings and leaves it to the developer to ascertain whether and how the functionality should be used.

Possible attributes are:

> REQUIRES_CUDA
>
> > Whether the new keyword implementation requires a functioning CUDA context in order to be functional. The default values is false. Example:
> >
> > > REQUIRES_CUDA=true
>
> UNIQUE_NAME

Whether the new keyword implementation needs Kappa Process to implement a unique name for each command queue instance using all of the arguments and map. The default is false, that either the name given in the statement is sufficient or that it is desirable for the same named command to be (re)used. Example:

```
UNIQUE_NAME=true
```

Kappa/Routine

Load a Kappa compiled subroutine or function from a shared library.

Possible syntax is:

```
!Kappa/Routine -> Load('routine','shared_library');
```

CUDAConfig

This will initialize the CUDA configuration namespace Value objects, if not already done by calling the kappa::Process.InitCUDAConfig method.

Possible syntax is:

```
!CUDAConfig;
```

Resource Relinquishing Keywords

Resource relinquishing statements free, unload, or relinquish access to a resource.

ContextReset

Reset a context, freeing resources such as modules, variables, values, etc. This command will execute after all command dependencies, including secondary and further dependencies, on the current context finish. It is a very good idea to still explicitly invoke the specific resource relinquishing statements. This statement sets a CANCELED completion status to cancel any further dependent commands (commands invoked after this statement that uses the same resources).

Possible syntax is:

!ContextReset -> *command_name*;

There are no current attributes for the ContextReset statement at this time.

FreeValue

Free a Value. This command will execute after all command dependencies, including secondary and further dependencies, on the Value have finished. It will explicitly delete a Value from a Kappa Namespace. This statement sets a CANCELED completion status to cancel any further dependent commands (commands invoked after this statement that use the Value).

Possible syntax is:

!FreeValue -> *value_name*;

There are no current attributes for the FreeValue statement at this time.

Free

Free a Variable. This command will execute after all command dependencies, including secondary and further dependencies, on the Variable have finished. It will explicitly delete a Variable from a Kappa Context. This statement sets a CANCELED completion status to cancel any further dependent commands (commands invoked after this statement that use the Variable).

Possible syntax is:

!Free -> *variable_name*;

There are no current attributes for the Free statement at this time.

C/ModuleUnload

Unload a C/Module. This command will execute after all command dependencies, including and further secondary dependencies, on the C/Module have finished. It will explicitly delete a C/Module from a Kappa Context. This statement sets a CANCELED completion status to cancel any further dependent commands (commands invoked after this statement that use the C/Module). Note that the shared library may still stay resident in memory depending on the loading mechanism.

Possible syntax is:

!C/ModuleUnload -> *module_name*;

There are no current attributes for the C/ModuleUnload statement at this time.

CUDA/ModuleUnload

Unload a CUDA/Module. This command will execute after all command dependencies, including secondary and further dependencies, on the CUDA/Module have finished. It will explicitly delete a CUDA/Module from a Kappa Context. This statement sets a CANCELED completion status to cancel any further dependent commands (commands invoked after this statement that use the CUDA/Module).

Possible syntax is:

!CUDA/ModuleUnload -> *module_name*;

There are no current attributes for the CUDA/ModuleUnload statement at this time.

Stop

Inform the Kappa background process command queue processor to start exiting. This background process will try to finish any running commands but no further commands on the queue will be accepted. This command will execute after all prior command dependencies, including secondary and further dependencies, have finished. This and the Finish statement ensure that commands are processed prior to program exit.

Possible syntax is:

> !Stop;

There are no current attributes for the Stop statement at this time.

Finish

Cause the Kappa Process thread execution to pause waiting for the background process to finish. This will shift thread scheduling priority to the background process. This command will execute after all prior command dependencies, including secondary and further dependencies, have finished. This and the Stop statement ensure that commands are processed prior to program exit.

Possible syntax is:

> !Finish;

There are no current attributes for the Finish statement at this time.

PopContexts

Try to pop (and free) prior Kappa Context objects until the requested context name is current. This command will execute after all command dependencies, including secondary and further dependencies, on the Context have finished. f the requested context name can not found, then nothing happens. It will explicitly delete all intervening Kappa Context objects and all of their resources (Variable objects, Module objects, etc.). The named context and its resources become current. This statement sets a CANCELED completion status to cancel any further dependent commands (commands invoked after this statement that use the PopContexts).

Possible syntax is:

> !PopContexts -> *context_name*;

There are no current attributes for the PopContexts statement at this time.

Processing Keywords

The processing statements launch C or CUDA kernels, invoke Kappa Process subroutines or functions, invoke callbacks to user functions, copy Variable objects, and invoke decision statements.

C/Kernel

Launch a C/Kernel.

Possible syntax is:

> !C/Kernel *attributes* -> *kernel_name@module_name(arguments) [map]*;
>
> !C/Kernel MODULE=*module_name attributes* -> *kernel_name(arguments)* [map];
>
> !C/Kernel -> *kernel_name@module_name(arguments) [map]*;
>
> !C/Kernel MODULE=*module_name* -> *kernel_name(arguments)* [map];

!C/Kernel *kernel_name@module_name(arguments) [map]*;

The *kernel_name* must match the 'extern “C”' kernel function name declared in the C module or a FUNCTION attribute must be given that specifies the 'extern “C”' kernel function name declared in the C module. The *module_name* must be the name of a previously loaded C/Module. The *arguments* and the *map* may be optional, depending on the kernel being launched.

Possible attributes are:

MODULE

The *module_name* must be the name of a previously loaded C/Module.

FUNCTION

The 'extern “C”' kernel function name declared in the C module. If the FUNCTION attribute is given, then the *kernel_name* may be any unique text string identifier desired that does not conflict with other C/Kernel FUNCTION or *kernel_name*'s. C/Kernel objects are assigned unique command names for command queuing that include their arguments and map.

The FUNCTION and *kernel_name* functionality may be further used to ensure unique command queue names and proper dependency processing. In other words, if you wish to launch multiple times the exact same kernel with the exact same arguments and map, use the FUNCTION attribute to specify the kernel function name and give a different *kernel_name* to each launch invocation.

Example:

FUNCTION=foobar

CUDA/Kernel

Launch a CUDA/Kernel.

Possible syntax is:

!CUDA/Kernel *attributes* -> *kernel_name@module_name(arguments) [map]*;

!CUDA/Kernel MODULE=*module_name attributes* -> *kernel_name(arguments)* [map];

!CUDA/Kernel -> *kernel_name@module_name(arguments) [map]*;

!CUDA/Kernel MODULE=*module_name* -> *kernel_name(arguments)* [map];

!CUDA/Kernel *kernel_name@module_name(arguments) [map]*;

The *kernel_name* must match the kernel name declared in the CUDA module or a FUNCTION attribute must be given that specifies the kernel name declared in the CUDA module. The *module_name* must be the name of a previously loaded CUDA/Module. The *arguments* and the *map* may be optional, depending on the kernel being launched.

Possible attributes are:

MODULE

The *module_name* must be the name of a previously loaded CUDA/Module.

FUNCTION

The kernel name declared in the CUDA module. If the FUNCTION attribute is given, then the *kernel_name* may be any unique text string identifier desired that does not conflict with other CUDA/Kernel FUNCTION or *kernel_name*'s. C/UDAkernels are assigned unique command names for command queuing that include their arguments and map.

The FUNCTION and *kernel_name* functionality may be further used to ensure unique command queue names and proper dependency processing. In other words, if you wish to launch multiple times the exact same kernel with the exact same arguments and map, use the FUNCTION attribute to specify the kernel function name and give a different *kernel_name* to each launch invocation.

Example:

FUNCTION=foobar

GRID

Specifies the grid for launching the CUDA kernel. See the CUDA Reference Manual cuLaunchGrid function for details. Specifying this attribute causes the GRIDX and GRIDY attributes to be ignored. Example:

GRID=[5, 8]

GRIDX

Specifies the grid width for launching the CUDA kernel. See the CUDA Reference Manual cuLaunchGrid function for details. Specifying the GRID attribute causes the GRIDX and GRIDY attributes to be ignored. Example:

GRIDX=5

GRIDY

Specifies the grid for launching the CUDA kernel. See the CUDA Reference Manual cuLaunchGrid function for details. Specifying the GRID attribute causes the GRIDX and GRIDY attributes to be ignored. Example:

GRIDY=8

BLOCKSHAPE

Specifies the block shape for launching the CUDA kernel. The default value, if a dimension is not specified, is one. See the CUDA Reference Manual cuFuncSetBlockShape function for details. Specifying the BLOCKSHAPE attribute causes the BLOCKSHAPEX, BLOCKSHAPEY, and BLOCKSHAPEZ attributes to be ignored. Example (letting the z dimension default to one):

BLOCKSHAPE=[16, 12]

BLOCKSHAPEX

Specifies the block shape x dimension for launching the CUDA kernel. The default value, if not specified, is one. See the CUDA Reference Manual cuFuncSetBlockShape function for details. Specifying the BLOCKSHAPE attribute causes the BLOCKSHAPEX, BLOCKSHAPEY, and BLOCKSHAPEZ attributes to be ignored. Example:

```
BLOCKSHAPEX=16
```

BLOCKSHAPEY

Specifies the block shape y dimension for launching the CUDA kernel. The default value, if not specified, is one. See the CUDA Reference Manual cuFuncSetBlockShape function for details. Specifying the BLOCKSHAPE attribute causes the BLOCKSHAPEX, BLOCKSHAPEY, and BLOCKSHAPEZ attributes to be ignored. Example:

```
BLOCKSHAPEY=12
```

BLOCKSHAPEZ

Specifies the block shape z dimension for launching the CUDA kernel. The default value, if not specified, is one. See the CUDA Reference Manual cuFuncSetBlockShape function for details. Specifying the BLOCKSHAPE attribute causes the BLOCKSHAPEX, BLOCKSHAPEY, and BLOCKSHAPEZ attributes to be ignored. Example:

```
BLOCKSHAPEZ=1
```

SHAREDMEMORY

Specifies the amount of dynamic shared memory available to each thread block when launching the CUDA kernel. The default value, if not specified, is zero. See the CUDA Reference Manual cuFuncSetSharedSize function for details. Example:

```
SHAREDMEMORY=( 12 * #BLOCK_SIZE * %sizeof{float} )
```

CACHE

Sets the preferred cache type for the CUDA kernel. Set this to CU_FUNC_CACHE_PREFER_SHARED to prefer more shared memory and set this to CU_FUNC_CACHE_PREFER_L1 to prefer more L1 cache.

```
CACHE=%CUDA{CU_FUNC_CACHE_PREFER_SHARED}
```

Please see the description of the cuFuncSetCacheConfig function in the CUDA Reference Manual for a description of setting the cache preference.

STREAM

The name of a stream to associate with the Kernel for doing concurrent kernel launches. The Kappa Stream object and CUDA stream and any necessary CUDA events are automatically created and managed. Example:

```
STREAM=mystream
```

Subroutine

Execute a previously loaded or defined Kappa Process subroutine.

Possible syntax is:

> !Subroutine *attributes* -> *subroutine_name*;
>
> !Subroutine -> *subroutine_name*;
>
> !Subroutine *subroutine_name*;

The *subroutine_name* must match the subroutine name previously loaded or defined in a Kappa Process subroutine. See the section Subroutine and Function Definition for how to define subroutines and functions and see this section for how to output and load subroutines. Since subroutines operate in the namespace context they are invoked at, they effect Value objects and Variable objects as the subroutine's statements execute. In this sense, subroutines are not “atomic” with respect to the completion and cancellation of their statements and their effects on non subroutine Value objects and Variable objects.

Completion and cancellation of statements occurs throughout subroutine and function invocations. It is possible and normal for some statements within a subroutine or function to execute to completion while others are canceled or canceled prior to invocation.

Possible attributes are:

UNROLL

> If set to false (the default), then subroutine loops are dynamic—there is a Namespace loop Value that is dynamically used for looping. If set to true, then the subroutine is unrolled as a macro for the number of times set by the LOOP attribute.

LOOP

> If the UNROLL attribute is given and true, then this specifies how many times to expand the subroutine as if it is a macro—the default is to expand it once.
>
> If the UNROLL attribute is not given or is false, then this sets the initial loop count for the subroutine. The default value is one. This is set into a Kappa Namespace Value with the name: "/kappa/routine/*subroutine_name*/loop". The subroutine's statements may modify this Value to lengthen or shorten the number of execution loops performed. At the end of the execution of the subroutine statements, the current value of this Kappa Namespace Value is decremented by one and then, if the Value is still greater than zero, the subroutine's statements are executed again.

Function

Execute a previously loaded or defined Kappa Process function.

Possible syntax is:

> !Function *attributes* -> *function_name(arguments) [map]*;
>
> !Function *attributes* -> *function_name*;

!Function -> *function_name(arguments) [map]*;

!Function -> *function_name*;

!Function *function_name(arguments) [map]*;

!Function *function_name*;

The *function_name* must match the function name previously loaded or defined in a Kappa Process function. See the section Subroutine and Function Definition for how to define subroutines and functions and see this section for how to output and load functions.

Function arguments are copied on input and output as appropriate (depending on whether the argument is an input, output, or both). Literal values, such as a text string or a number, may only be inputs. Value objects and Variable objects may be both inputs and outputs. Since the function works on local copies of the Value objects and Variable objects, if the statements in the function cancel midway through the function, the external (to the function) Value objects and Variable objects are not effected. In this sense, functions are "atomic" with respect to the completion and cancellation of their statements and their effects on non function Value objects and Variable objects.

Completion and cancellation of statements occurs throughout subroutine and function invocations. It is possible and normal for some statements within a subroutine or function to execute to completion while others are canceled or canceled prior to invocation.

There are no current attributes for the Function statement at this time.

GetNullMask

Copy a Variable's NullMask into a separate Variable.

Possible syntax is:

!GetNullMask *attributes* -> *new_variable_name* = *previous_variable_name*;

!GetNullMask -> *new_variable_name* = *previous_variable_name*;

! GetNullMask *new_variable_name* = *previous_variable_name*;

There are no current attributes for the GetNullMask statement at this time.

SetNullMask

Use a NullMask (two dimensional) Variable to over write a Variable's NullMask.

Possible syntax is:

!SetNullMask *attributes* -> *variable_name* = null_mask_*variable_name*;

!SetNullMask -> *variable_name* = null_mask_*variable_name*;

! setNullMask *variable_name* = null_mask_*variable_name*;

There are no current attributes for the SetNullMask statement at this time.

CopyVariable

Copy one Variable to another Variable.

Possible syntax is:

!CopyVariable *attributes* -> *new_variable_name* = *previous_variable_name*;

!CopyVariable -> *new_variable_name* = *previous_variable_name*;

!CopyVariable *new_variable_name* = *previous_variable_name*;

There are no current attributes for the CopyVariable statement at this time.

Decision

Decide whether to continue or cancel execution for the dependency arguments given.

Possible syntax is:

? logical_expression -> arguments;

The arguments for a decision statement are a list of Variable objects and Value objects to make dependent on the decision. The logical_expression must only contain numbers, configuration values, Value objects, and logical or numeric operators. All configuration values and Value objects must be resolvable to numeric or boolean values. The logical_expression is evaluated and, if it is zero, the decision statement reports a CANCELED completion status. This cancels execution of all statements that are dependent on the Variable objects or Value objects in the arguments list. It reports a FINISHED completion status if the logical_expression evaluates to nonzero. It reports a FAILED completion status (which will have the same effect as a CANCELED completion status) if the logical_expression can not be evaluated to an integer or boolean value (for example if configuration values or Value objects can not resolve or that resolve to text values that are not convertible to numeric values).

For boolean values for attributes or Value objects, "true" or "on" and "false" and "off" or numeric values may be given. For Indices Value objects, a comma delimited list of integer values with optional spaces may be given. For logical and numeric expressions, configuration values and Value objects may be used, as long as they are resolvable. For logical and numeric expressions, parenthesis are supported and outer, enclosing parenthesis may be mandatory for proper classification. For logical and numeric expressions, the following binary operators are currently supported:

+ - * / ^ or and xor < > == <= >= !=

IO

This executes an IOCallback if one is found that is registered with the same *callback_name* as given in the IO statement. See the IOCallbacks section for further information on using the IOCallback functionality.

Possible syntax is:

!IO -> *callback_name*(*variable_name*) [*map*];

The *map* should be one of:

- Input:

  [= *variable_name*]
- Output:

  [*variable_name* =]
- Input and Output:

  [*variable_name*]

  or

  [*variable_name* = *variable_name*]

There are no current attributes for the IO statement at this time.

Informational Keywords

Print

Print values or Value objects to standard output.

Possible syntax is:

    !Print (*arguments*);

arguments is a list of values or Value objects to print.

There are no current attributes for the Print statement at this time.

Timer

The statement:

    !Timer -> <*timer_name*>;

will, the first time it is given, create a timer. The subsequent times it it is given, it will output to standard error the elapsed time since the timer was last invoked:

```
Processing time: 42.984 (ms)
```

The times may include scheduling delays, system overhead and delays, etc.

Context

The statement:

    !Context -> <*context_name*>;

will, if if is subsequently repeated in the same Kappa Process, report to standard error the device memory usage:

```
Device: Starting Free Memory: 514719744
```

```
        Ending:  Free Memory: 514719744
          Difference Memory: 0
                    Total: 536674304
                     Used: 146748
```

When the "!Context" statement with the same context name is given (repeated), it is given a "final" dependency status. This means that it is deferred for execution until all other dependencies on that named context have finished.

DisplayAll

Displays, to standard output, the GPU properties and attributes.

Possible syntax is:

!DisplayAll;

CUDA/Kernel/Attributes

The attributes of a kernel, as it is actually compiled and loaded on the GPU to execute, can be retrieved by executing the statement:

!CUDA/Kernel/Attributes MODULE=<*module_name*> -> <*kernel_name*>;

where <*module_name*> is the module name given and <*kernel_name*> is the name of the kernel. This create the following Value objects:

/kappa/CUDA/<*module_name*>/<*kernel_name*>#MaxThreadsPerBlock

/kappa/CUDA/<*module_name*>/<*kernel_name*>#StaticSharedMemory

/kappa/CUDA/<*module_name*>/<*kernel_name*>#RegistersPerThread

/kappa/CUDA/<*module_name*>/<*kernel_name*>#ConstantMemory

/kappa/CUDA/<*module_name*>/<*kernel_name*>#ThreadLocalMemory

/kappa/CUDA/<*module_name*>/<*kernel_name*>#PTXVersion

/kappa/CUDA/<*module_name*>/<*kernel_name*>#BinaryVersion

Please refer to the CUDA Reference Manual or Programming Guide for further details on these attributes and their uses.

SQL Keyword

The SQL keyword currently has:

- connect
- disconnect
- begin
- commit

- rollback
- select
- read
- write
- execute

commands.

The SQL keyword can use the KappaAPRDBD database driver to use the Apache Portable Runtime (APR) database drivers. The KappaAPRDBD database driver is a subclass of the kappa::DBD class. Other database drivers may be used by properly subclassing the kappa::DBD class contained in kappa/DBD.h.

SQL keyword operations are kept in dependency order by database handle identifier (*dbhandle*). This means that commands on the same *dbhandle* are sequential. To have multiple, simultaneous, asynchronous database operations, use separate database handle identifiers.

SQL statements for the SQL keyword may consist of single or double quoted strings embedded in the scheduling script or may be retrieved from configuration files as configuration values. Currently, multi-line quoted strings are not supported in the scheduling scripts and must instead be placed in configuration files.

It may be helpful to refer to the Example usage of the SQL and Expand keywords.section while reading the following sections.

Format strings

Format strings are used to specify the binary data structure and, if necessary, any padding needed for reading and writing from Variables and the database. They are similar to the format strings used in the Apache Portable Runtime which, in turn, are similar to the format strings for the scanf function. The format will specify the data conversion to or from the database connection. For calculating the size taken in the binary format in the Variable, the sizes are rounded up to the next padding size of a multiple of 4.

The following are supported:

- The '*' assignment-suppression character (for specifying structure padding).
- The decimal integer which, in this case specifies the width, in bytes, for either padding or variable length fields such as character strings. Note that, for assignment-suppression padding, this number overrides any other formatting characters. If this number is larger that the size, in bytes, of the conversion specified type, then this number, rounded up to the next multiple of 4 bytes, is used.
- The optional type modifier character, 'h', indicates that the conversion will be one of **d** or **u** and will correspond to a short int or unsigned short int.
- The optional type modifier characters, 'hh', indicates that the conversion will be one of **d** or **u** and will correspond to a signed char or unsigned char.
- The optional type modifier character, 'l', indicates that the conversion will be one of **d**, **u**, or **f** and will correspond to a long int, unsigned long int, or double.
- The optional type modifier characters, 'll', indicates that the conversion will be one of **d** or **u** and will correspond to a long long or unsigned long long.
- The conversion specifier, '%', that specifies a literal '%' character by using two of them in sequence: '%%'.
- The conversion specifier, 'd', for a corresponding signed integer.
- The conversion specifier, 'u', for a corresponding unsigned integer.
- The conversion specifier, 'f', for a corresponding float.
- The conversion specifier, 's', for a character array--be sure to specify a preceding decimal integer for the size of the character array.
- The conversion specifier, 'p' for an extended type. The next character must be a 'D'. The character following the 'D' character specifies the extended type conversion as follows:
    - The extended conversion specifier, 't', will correspond to text (converted to a character array--be sure to specify a preceding decimal integer for the size of the character array).
    - The extended conversion specifier, 't', will correspond to text (converted to a character array--be sure to specify a preceding decimal integer for the size of the character array).

- The extended conversion specifier, 'i', will correspond to a time (converted to a character array--be sure to specify a preceding decimal integer for the size of the character array).
- The extended conversion specifier, 'd', will correspond to a date (converted to a character array--be sure to specify a preceding decimal integer for the size of the character array).
- The extended conversion specifier, 'a', will correspond to a datetime (converted to a character array--be sure to specify a preceding decimal integer for the size of the character array).
- The extended conversion specifier, 's', will correspond to a ztimestamp (converted to a character array--be sure to specify a preceding decimal integer for the size of the character array).
- The extended conversion specifier, 'z', will correspond to text (converted to a character array--be sure to specify a preceding decimal integer for the size of the character array).
- The extended conversion specifier, 'b', will correspond to a BLOB (not currently converted).
- The extended conversion specifier, 'c', will correspond to a CLOB (not currently converted)
- The extended conversion specifier, 'n', will correspond to a pointer (not currently converted)

Meta specifiers

In addition, there are the following 'meta' specifiers:

'='	The following specifier is for a primary key id.
'+'	The following specifiers are for measures.
'-'	The remaining specifiers are for parameters.

If the format is for a STAR format operation, then the first format specifiers specify field conversions for dimension fields. The '+' specifier, designates that the specifiers that follow it are for measure fields. The '-' specifier, designates that the specifiers that follow it are not for either dimension or measure fields.

The '=' specifier, marks the next field as a field for a primary key id. In operations that support it, these primary key values are placed in separate Variables.

Assuming an example query where:

- the first field is an (unsigned) integer primary key id,
- the next three fields are unsigned integers that are (STAR format) dimension fields,
- that a 4 byte padding is required next,
- and then a measure field that is a double

gives the following format string:

- =%u %u %u %u %*4u +%lf

with the following Variable's data structure:

- struct EXAMPLE_STRUCT {
  - unsigned dimension1;
  - unsigned dimension2;
  - unsigned dimension3;
  - /* 4 byte padding */
  - double measure;
  - };

The integer primary key id field is not stored in this Variable's data structure. There would be a separate key Variable data structure similar to the following:

- struct EXAMPLE_KEY_STRUCT {
  - unsigned primary_key;
  - };

Conversions

Format specifiers will convert the database field to and from the data type requested, if possible. The format specifiers correspond to the binary data structure desired in the Variable--not necessarily to the data type of the database field.

Aligntest

There is an example program, aligntest, included in the Kappa share/extras for checking the alignment and padding of data structures on the host and GPU. See the INSTALL file for compilation instructions. Modify the aligntest.h, main.cpp, and aligntest.cu files as appropriate to test the data structure you wish to learn about. As distributed, the program puts out the following:

```
Offset 0: 4
Offset 1: 4
Offset 2: 4
Offset 3: 4
Offset 4: 4
Offset 5: 4
Offset 6: 4
Offset 7: 4
Offset 8: 8
Alignment mismatch 0: 0
Alignment mismatch 1: 0
Alignment mismatch 2: 0
Alignment mismatch 3: 0
Alignment mismatch 4: 0
Alignment mismatch 5: 0
Alignment mismatch 6: 0
Alignment mismatch 7: 0
Alignment mismatch 8: 0
```

Showing that the first six dimension unsigned structure fields (dima - dimf) have a size of 4 bytes and no padding, the unsigned measurea and float measureb have a size of 4 bytes each and no padding, and the double measurec has a size of 8 bytes and no padding to the end of the structure. This also shows (in this example output) that there are no difference between this structure on a 64 bit host system and the code on a GPU compiled for 32 bits.

This is also a good, simple example of the usage of the IO keyword and callback mechanism.

SQL keyword connect command

!SQL -> connect@*dbhandle*(*driver_name*,*DBPARAMS*);

The *dbhandle* identifies the database connection. SQL keyword operations are kept in dependency order by *dbhandle*--commands on the same *dbhandle* are sequential.

The *driver_name* is usually one of: pgsql, freetds, mysql, odbc, sqlite, or oracle.

On Linux, the RPM package names and descriptions are:

apr-util-freetds.x86_64 : APR utility library FreeTDS DBD driver

apr-util-mysql.x86_64 : APR utility library MySQL DBD driver

apr-util-odbc.x86_64 : APR utility library ODBC DBD driver

apr-util-pgsql.x86_64 : APR utility library PostgreSQL DBD driver

apr-util-sqlite.x86_64 : APR utility library SQLite DBD driver

apr-util-oracle is not generally available in distributions but may be built using the apr-util source code.

On Windows, the following are installed with Kappa:

apr_dbd_mysql-1.dll

apr_dbd_odbc-1.dll

apr_dbd_oracle-1.dll

apr_dbd_pgsql-1.dll

apr_dbd_sqlite3-1.dll

The *DBPARAMS* are:

- FreeTDS (for MSSQL and SyBase)

  username, password, appname, dbname, host, charset, lang, server

  each followed by an equal sign and a value. Such key/value pairs can be delimited by white space, semicolon, vertical bar or comma.

- MySQL

  host, port, user, pass, dbname, sock, flags, fldsz, group, reconnect

  each followed by an equal sign and a value. Such key/value pairs can be delimited by white space, semicolon, vertical bar or comma.

- ODBC

  datasource, user, password, connect, ctimeout, stimeout, access, txmode, bufsize

  each followed by an equal sign and a value. Such key/value pairs can be delimited by white space, semicolon, vertical bar or comma.

- Oracle

  user, pass, dbname, server

  each followed by an equal sign and a value. Such key/value pairs can be delimited by white space, semicolon, vertical bar or comma.

- PostgreSQL

  The connection string is passed straight through to Pqconnectdb similar to the following:

  host=myhost port=5432 dbname=mydb user=mypguser password=mypwd

- SQLite3

  The connection string is passed straight through to sqlite3_open.

SQL keyword disconnect, begin, commit, and rollback commands

!SQL -> disconnect@*dbhandle*();

!SQL -> begin@*dbhandle*();

!SQL -> commit@*dbhandle*();

!SQL IF_FAIL=true IF_CANCEL=true -> rollback@*dbhandle*();

By default the database connection defaults to the transaction control of the underlying database driver. For most drivers, this defaults to immediate transaction commits. To have an explicit, multiple command transaction, use the begin, commit, and rollback commands. A transaction includes the commands between a begin command and the next commit, rollback, or disconnect. A commit or rollback may cause the transaction mode to switch back to the default transaction mode until another begin command is given.

An implicit or explicit disconnect command, if proceeded by an open begin command, defaults to an implicit rollback. This means that, because commands are dependent on the *dbhandle* and that dependent commands are canceled on the failure of a command, that:

- a begin commmand,
- followed by a series of transaction commands,
- a commit command,
- and then (eventually) a disconnect command

(all on the same *dbhandle*) causes the usual desired behavior of having the transaction commit

if all the commands are successfully executed and to have the transaction rollback if any fail.

SQL keyword select command

!SQL -> select@*dbhandle*(*sql_select_statement*, *format_string*, *#num_rows*, *#num_cols*,

#row_size);

!SQL -> select@*dbhandle*(*sql_select_statement_with_formatting*, *index*, *Categories_Variable*, *format_string*, *#num_rows*, *#num_cols*, *#row_size*);

!SQL -> select@*dbhandle*(*sql_select_statement_with_formatting*, *index*, *Categories_Variable*, *Categories_Variable_Key*, *format_string*, *#num_rows*, *#num_cols*, *#row_size*);

The *sql_select_statement* or *sql_select_statement_with_formatting* are a SQL select or other SQL statement that results in a record set. If there is formatting in the statement, it is to specify the parameter binding for data contained in the *Categories_Variable* or (if a primary key format specification is given) the *Categories_Variable_Key*. The *index* is the row offset to use for the *Categories_Variable* and/or *Categories_Variable_Key* Variables when binding parameters for the statement.

The *format_string* specifies the conversion and formatting that will occur when reading data.

The *#num_rows*, *#num_cols*, and *#row_size* are Namespace Value names for the number of rows readable from this statement, the number of columns, and the row size (in bytes) respectively.

Possible attributes are:

ASYNC

If set to false (the default), then the statement is executed synchronously. If set to true, then the statement is executed on another host thread.

STAR

If the STAR attribute is given and true, then STAR database format processing is enabled. STAR database format processing converts dimension character fields that are not a form of integer to a corresponding integer translation value (specify a key Variable to record and keep the association between the integer value for a translated value that is usable on a GPU and the original value still in the database). Dimensions and measures are tracked separately.

RANDOM

If set to true (the default), then the cursor allows random access scrolling. If set to false, then the cursor is not set for random access scrolling and will work for database drivers that do not support random access cursor scrolling.

SQL keyword read command

!SQL -> read@*dbhandle*(*Variable*,*#max_rows_to_read*,*#rows_read*);

!SQL -> read@*dbhandle*(*Variable*, *Keys_Variable*, *#max_rows_to_read*, *#rows_read*);

The *Variable* is the name of the Variable to copy the data into. The *Keys_Variable* is the name of the Variable to copy the primary key id data into. The keys are useful for keeping the association between translated dimension field values and the corresponding record in the database. The keys are also useful for fast SQL update and delete operations using the SQL keyword write command or for selecting specific rows using the SQL keyword select command with bound Variables.

The #*max_rows_to_read* is the (maximum) number of rows to read in this operation. The #*rows_read* is the Namespace Value name for the number of rows actually read in this operation.

Numeric integer NULL values are read as zero, float or double NULL values are read as quiet NaN (quiet_NaN), and any NULL values that convert to character arrays convert to empty character arrays.

Possible attributes are:

ASYNC

If set to false (the default), then the statement is executed synchronously. If set to true, then the statement is executed on another host thread.

FAST

If the FAST attribute is given and true, then the Variable is not cleared with zeroes prior to the database read. If there are character arrays or NULL fields, then parts of these content locations in the Variable may be the previous content.

SQL keyword write command

!SQL -> write@*dbhandle*(*sql_statement_with_formatting*,*Output_Variable*, *Output_Keys_Variable*, #*rows_to_write*, #*rows_affected*);

!SQL -> write@*dbhandle*(*sql_statement_with_formatting*,*Output_Variable*, #*rows_to_write*, #*rows_affected*);

The *sql_statement_with_formatting* is a SQL statement with formatting in the statement, that specifies the parameter binding for data contained in the *Output_Variable* or (if a primary key format specification is given) the *Output_Variable_Key*. Usually the SQL statement would modify the database in some way but this command can be used for any SQL statement that needs to bind parameters but does not return a record set to be read.

The #*rows_to_write* is the (maximum) number of rows of the *Output_Variable* or (if a primary key format specification is given) the *Output_Variable_Key* to bind to the parameters in the SQL statement and execute in this operation. The #*rows_affected* is the Namespace Value name for the (total) number of database rows actually affected in this operation.

Possible attributes are:

ASYNC

If set to false (the default), then the statement is executed synchronously. If set to true, then the statement is executed on another host thread.

SQL keyword execute command

!SQL -> execute@*dbhandle*(*sql_statement*,#rows_affected);

The *sql_statement* is a SQL statement to execute. No parameter binding is done and it is assumed that there is no resulting record set.

The #*rows_affected* is the Namespace Value name for the (total) number of database rows actually affected in this operation.

Possible attributes are:

ASYNC

If set to false (the default), then the statement is executed synchronously. If set to true, then the statement is executed on another host thread.

Synchronize Keyword

!Synchronize name(*args*)

The Synchronize command halts preparation of instructions at the point it is encountered until the Process Notification Finish for it is encountered. This will make the foreground preparation of instructions stop until the Synchronization command is executed in the background—which will not happen until its dependencies are done.

Note that CUDA/Kernel GRID, BLOCKSHAPE, SHAREDMEMORY, and others that are not listed below do NOT need a !Synchronize; command--they are resolved on the background thread.

Note that Value can retrieve an Indice or a single number from a Variable. This means that an Indice or number can come from any source, GPU or CPU calculation, SQL query, etc.

The following are resolved at prepare time and need a !Synchronize (); command if they use a calculated Value (instead of a literal string, number, or truth value):

- Any keyword that uses the: MODULE or STREAM attributes
- Keyword Expand's arguments
- Keyword Subroutine's LOOP and UNROLL attributes
- Any keyword that REQUIRES_CUDA, needs a UNIQUE_NAME, and has a module file argument
- Keyword Context's CCUDA_CONTEXT_TYPE, CONTEXT_FLAGS, USE_CURRENT, and REPORT_MEMORY attributes
- Keyword CUDA/Module's JITMaxRegisters, JITThreadsPerBlock, JITRecordCompileTime, JITOptimizationLevel, JITTargetFromContext, JITTarget, JITFallback, and PRINTF_FIFO_SIZE attributes and its module file argument
- Keyword C/Module's module file argument
- Keyword Variable's VARIABLE_TYPE, STREAM, and DEVICEMEMSET attributes
- Keyword CopyVariable's from variable name argument
- Keyword IO's variable name argument
- Keyword C/Kernel's FUNCTION attribute
- Keyword CUDA/Kernel's FUNCTION and OUTPUT_WARNING attributes

- Keyword CUDA/Module/Variable's variable name argument
- Keyword Array's WIDTH, HEIGHT, ARRAY_FORMAT, NUMBER_CHANNELS, and DEPTH attributes and variable name argument
- Keyword Texture's ADDRESS_MODE0, ADDRESS_MODE1, FILTER_MODE, and TEXTURE_FLAGS attributes and array name argument
- Keyword Value's FROM_FLOAT attribute

Expand Keyword

Assuming a subroutine with a labels attribute (and maybe a labelset attribute) similar to the following:

```
<kappa subroutine=mysub labels='$label'>
        !Print ('label_$label  ', #value_$label );
</kappa>
```

along with a statement:

!Expand -> mysub(*label_limits*);

allows replacing, in the subroutine, any occurrence of $label with 0..#number

where #number is the corresponding label limit argument in the Expand statement (use a Synchronize statement if the #number is calculated):

```
!Synchronize (#number);
```

The Expand statement allows creating subroutines as (tensor) indexed components that can be expanded dynamically at runtime. This gives a real, practical implementation that allows for true concurrent kernel execution and algorithm step sizing. Properly used, this allows maximum occupancy and use of GPU and CPU.

Labels can be placed in attribute values (such as stream ids), module names, kernel names, Value names, kernel arguments, and Variable names–among other places. These labels are then expanded (at runtime) with numeric ranges or Indices. Numeric ranges and Value Indices can be sliced from Variables using prior Kappa Library features. This allows for automatic parallelism and sizing using runtime data in a natural (tensor) index component manner. These labels can be used to create parallel execution dependency streams, vary across GPU or CPU, select/split/slice datasets for parallelism, select kernels, perform data parallel combinatoric expansions, etc.

The *label_limits* are arguments that are either unsigned integers or Indices. The number of arguments must match the number of labels given. If an argument to Expand is an unsigned integer, *label_limit*, then the label $label is replaced with 0 .. *label_limit*. If the argument is a Value Indices, then it is replaced with each value in the Indices. In either case, there is a separate expansion of the subroutine for each label replacement except for cases where a label occurs in the Value for a Subroutine statement LOOP attribute.

In the case where a label occurs in the Value for a Subroutine statement LOOP attribute, the label substitution that occurred in the LOOP Value is also used for the same label in the subroutine.

If a subroutine with a 'labels' attribute is called with !Subroutine; as the top level call, then each label is replaced with a literal zero (there is always a default action for any subroutine with labels).

Possible attributes are:

LABELSET

A labelset attribute may be given when defining a subroutine and the corresponding LABELSET attribute should be used for the Expand statement. Only subroutine labels with the matching labelset are expanded using the arguments from the Expand statement—default expansions are done for other labelset labels. Using a labelset is a good idea for anything that might become part of a library. This prevents labels from conflicting between libraries and user code.

Example usage of the SQL and Expand keywords.

The following example shows the combined usage of the SQL and the Expand keywords to dynamically size and run in parallel a task to retrieve data from a SQL data source for processing by a GPU. This example assumes a database table in standard star format named STAR_TABLE that has a a field, cat_pk_sid, that is usable for splitting the processing into parallel operations. This field would generally have a foreign key relationship to a master table that defines the permissible values for this field.

This example consists of two Kappa subroutines: *sqlio* and *sqlprocess*. The subroutine *sqlio* is unrolled within the *sqlprocess* subroutine using the Subroutine statement. The subroutine *sqlprocess* is expanded in the main Kappa scheduling script—it also expands the labels in the *sqlio* subroutine.

The SQL keyword read commands in the *sqlio* subroutine (and their corresponding select commands in the *sqlprocess* subroutine) are executed asynchronously. The CUDA/Kernel launches in the *sqlio* subroutine use the same stream identifier as the corresponding Variable creation statements in the *sqlprocess* subroutine and so they execute on the same CUDA streams as the Variables use for data transfer. Since the streams are expanded, the data transfers are overlapping with other data transfers and kernel launches and, if a suitable (GF100) GPU is being used, the kernel launches give concurrent kernel execution.

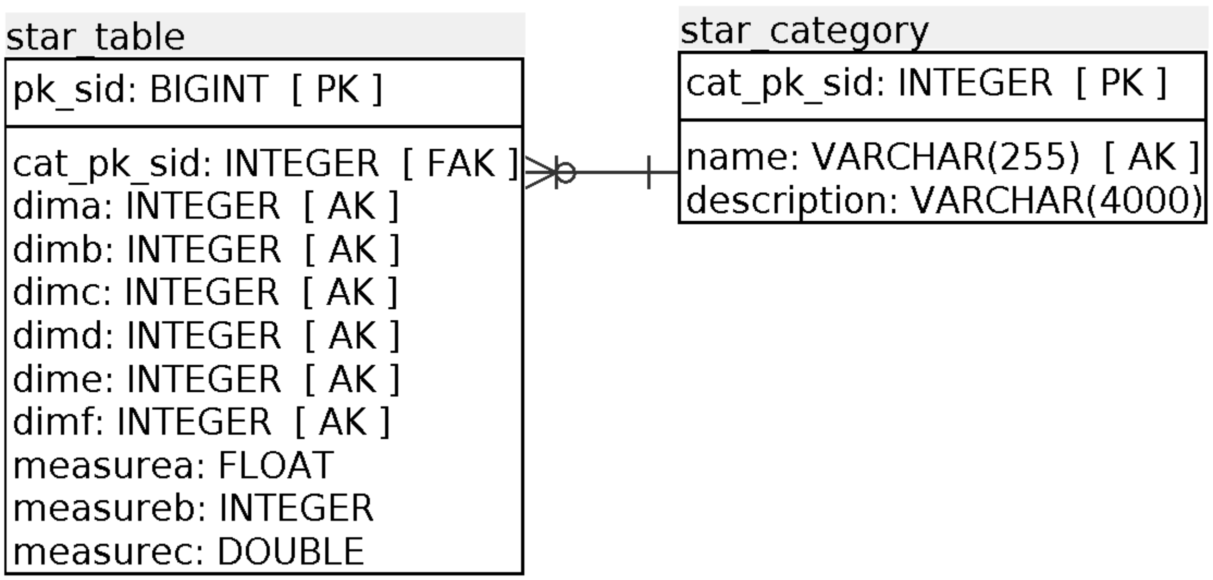
The SQL operations on the *dbhandle_$a* are expanded and so, if they have an ASYNC=true attribute, run asynchronously in parallel.

This example is able to execute the SQL operations in parallel and the CUDA kernels concurrently at very high speed on commodity multi-core CPU and GF100 hardware.

```
<kappa subroutine=sqlio labels='$a' labelset='sql'>
// The main IO loop
        !SQL ASYNC=true FAST=true -> read@dbhandle_$a(OUT_$a, #chunk_size, #rows_read_$a);
        !CUDA/Kernel STREAM=str_$a OUTPUT_WARNING=false -> sqltest@sqltest(OUT_$a,
        #rows_read_$a) [ = OUT_$a #rows_read_$a];
</kappa>


<kappa subroutine=sqlprocess labels='$a' labelset='sql'>
        !SQL -> connect@dbhandle_$a('pgsql',{PGPARAMS});
        !SQL ASYNC=true STAR=true -> select@dbhandle_$a('select pk_sid, dima, dimb, dimc, dimd,
        dime, dimf, measurea, measureb, measurec from star_table where cat_pk_sid= %u order by
        dima;', $a, Categories, '=%lu %u %u %u %u %u %u +%f %u %lf', #num_rows_$a,
        #num_cols_$a, #row_size_$a);


// Get the number of rows to process at once using an if evaluation.
        !Value -> rows_allocate_$a = if ( ( #chunk_size < #num_rows_$a ) , #chunk_size ,
        #num_rows_$a );
        !Variable STREAM=str_$a VARIABLE_TYPE=%KAPPA{LocalToDevice} ->
        OUT_$a(#rows_allocate_$a, #row_size_$a);


// Calculate how many iterations based on the number of rows and
// how many rows to process at once.
        !Value -> numloops_$a = ( #num_rows_$a / #chunk_size );


// Perform a synchronization so the #numloops_$a Value is ready
        !Synchronize (#numloops_$a);
        !Print ('number of loops: ', #numloops_$a, ' = ' , #num_rows_$a, ' / ' , #chunk_size );


        !Subroutine LABELSET='sql' UNROLL=true LOOP=#numloops_$a -> sqlio;
        !SQL -> disconnect@dbhandle_$a(); // disconnect dbhandle
</kappa>
```

```
<kappa>
// Setup the CUDA context and load the CUDA module
        !Context ->  context;
        !CUDA/Module -> sqltest = 'sqltest/sqltest.cu';

//Set the size of the data to process at once
        !Value -> chunk_size = 65536;

// Connect to the database and get the categories to use for splitting into parallel processes
        !SQL -> connect@dbmain('pgsql',{PGPARAMS});
        !SQL -> select@dbmain('select distinct cat_pk_sid from star_table;', '%u', #num_rows_cat,
        #num_cols_cat, #row_size_cat);
        !Variable -> Categories(#num_rows_cat,#row_size_cat);
        !SQL -> read@dbmain(Categories,#num_rows_cat,#rows_read_cat);
        !SQL -> disconnect@dbmain();

        !Value -> cat_indice = Categories;
        !Print ( 'number of categories: ', #rows_read_cat, 'categories: ', #cat_indice);
// Synchronize the Value of how many categories so that Expand can use it as an argument
        !Synchronize (#rows_read_cat);

// Expand and run the processing in parallel across the categories
        !Expand LABELSET=sql -> sqlprocess(#rows_read_cat);

// Unload, cleanup, stop
        !CUDA/ModuleUnload -> sqltest;
        !ContextReset -> Context_reset;
        !Stop;
        !Finish;
</kappa>
```

The database schema for the previous example is shown in this image:



with a data structure similar to the following:

```
typedef struct {
  unsigned dima;
  unsigned dimb;
  unsigned dimc;
  unsigned dimd;
  unsigned dime;
  unsigned dimf;
  float measurea;
  unsigned measureb;
  double measurec;
} TEST_STRUCT;
```

and with contents in the star_category table of the following:

```
 cat_pk_sid | name |   description
------------+------+-----------------
          1 | CATA | First category
          2 | CATB | Second category
          3 | CATC | Third category
          4 | CATD | Fourth category
```

Example program (Kappa Interpreter and "Compiler"): ikappa

Kappa comes with an example command line program to use for developing Kappa Process processes. It can be used interactively or with batch files for testing. Running it with the "-h" option makes it display its usage:

```
$ ikappa -h

Usage is: ikappa [options] [kappa_file.k ...]

The form of the options is: [-achilkns] [-d device# ] [-o output_directory] [-m module_file] [-f
routine_function] [-e routine_name]

The options are:

    -a    Remember anonymous statements.
    -c    Process keyword config.
    -d    Use device number device#.
          (This option may be given multiple times.)
    -e    Specifies the subroutine name to load and execute from a library.
          (This option may be given multiple times.)
    -f    Specifies the function or subroutine name to load from a library.
          (This option may be given multiple times.)
    -i    Register IOCallback.
    -l    Set CUDA address space to 64 bit.  Assumes that '-m32' is not given to nvcc.
    -k    Add Keyword keyword.
    -n    Do not execute statements.
    -s    Do not read standard input if no file arguments are given.
    -o    Output routines to this output_directory.
    -m    Specifies the module library file for loading routines from.
    -h    Print this usage statement.
```

The statements may be ran on multiple GPUs by specifying the "-d" option for each GPU. The order of the specification of the "-d" option will select in what order the GPUs are used.

The "-a", "-n", and "-o" options are useful to process the statements from the Kappa Process statement files into C++ source code files in a project directory. Please note that the "-o" option will overwrite any files of the corresponding names. Created in the project directory is a file named "compile.sh" which contains the commands necessary to configure and compile a libKappaRoutines shared library on platforms that support the autoconf tools.

If no Kappa Process statement file is given, then it reads Kappa Process statements from the standard input. So, assuming there is a Kappa Process statement file named *test.k*, then the following two commands are different ways to execute that file:

```
$ ikappa test.k
$ ikappa < test.k
```

Multiple Kappa Process statement files may be given as long as only the last file contains the !Stop; and !Finish; statements.

Kappa Process Language Extension

The Kappa library is meant to be extended to meet the needs of the organizations and subject areas utilizing it. The primary goal of Kappa is to allow for the creation of sophisticated, powerful, and complex processing that retain simple and easy-to-use interfaces.

The Kappa Process language statements and instructions can have new Keyword commands defined and can aggregate increasingly complex processing using C and CUDA kernels into subroutines and functions that are loadable and callable as libraries.

Rather than have inputs and outputs to the the Kappa processing only available at the start and end as is traditional with functions and libraries, Kappa uses the callback mechanism using IOCallback functions and developer defined keywords to allow input and output to be called throughout processing. This makes it so that it can dynamically effect the sizing, scheduling, and behavior of the processing. This is especially important with modern parallel processing such as CUDA and OpenMP where the “start” and “end” of certain processing might only be deterministic by severely limiting the power of the processing.

kappa::Command::Keyword

A new keyword command can be added to Kappa by sub-classing the kappa::command::Keyword class. The kappa::command::Keyword class is in turn a sub-class of the kappa::Command class. The kappa::command::Keyword class and the kappa::Command class are setup to not allow copy or assignment.

In the shared library implementing the new keyword, there must be a factory function that instantiates the new kappa::command::Keyword sub-class and returns a pointer to it cast to a (void *) pointer. This factory function must be declared as 'extern “C”'.

A Kappa Process uses the program thread it is invoked on to parse, prepare for execution, and determine dependencies. It creates a separate, background process thread for scheduling and execution. This background process thread is the thread associated to a CUDA context—not the original program thread.

When creating a new keyword as a sub-class of kappa::command::Keyword, new implementations of the Prepare and Execute methods must be defined. These two methods reflect the dual execution nature of the Kappa library. The Prepare methods is called and executed on the program thread while the Execute method is called and executed on the background process thread. The Prepare method should initialize and validate its static data and access to data and determine dependencies. The

Execute method then performs the actions that need access to the CUDA context and resources or current runtime execution Value objects or other runtime current values. It is worth stating very explicitly: there may be a significant difference in time between when the Prepare and the Execute methods are called, they are called on different threads, and the program values will be significantly different at the different times that they are called. The Prepare method is always called prior to the Execute method and it is possible the Execute method is never called if the command is canceled.

From its constructor or Prepare method, the kappa::command::Keyword sub-class is responsible for either initializing the Command parent class with the name or calling the kappa::Command.SetName method to set the name of the command. From its Execute method, it is responsible for calling the appropriate kappa::Command::SetCanceled, kappa::Command::SetFailed, kappa::Command::SetFinished and kappa::Command::Notification methods. It may override the kappa::Command::UpdateStatus method to check for and update its status if it is running in some asynchronous mode and it is then responsible for calling the appropriate kappa::Command::SetCanceled, kappa::Command::SetFailed, kappa::Command::SetFinished and kappa::Command::Notification methods.

The kappa::Process.RegisterKeywordCommandData method may be used to set the data to be passed to the Prepare method. A common use for this would be to pass a data structure or class object instance which contains everything the new keyword would need to access what it needs to function properly.

kappa::command::KeywordCSV example

The KeywordCSV.h and KeywordCSV.cpp create a new CSV keyword and are an example that illustrates the previous discussion and provides a commented example. It provides examples and has comments discussing:

- defining a factory function,
- what data is available at what stages,
- how and when to Prepare and Resolve attributes,
- how and when to Prepare and Resolve arguments,
- how to work with resource dependencies,
- how and when to access CUDA context resources,
- how to create Variable objects and Value objects,
- and examples of how to set command completion status and notification.

IOCallback

The kappa::Process.RegisterIOCallback method may be used to register an IOCallback of a given name with a callback function and user data pointer. The name must match the name given in the Kappa Process statement:

!IO -> *name*(*argument*) [*map*];

The *argument* is the name of a Variable and the *map* is the appropriate map to specify the input and output dependencies:

- Input: [= *argument*]
- Output: [*argument* =]
- Input and Output: [*argument* = *argument*] or: [*argument*]

The callback function is called on the background process thread that is associated with a CUDA context. The developer must ensure proper thread-safe locking occurs around access to the user_data (Hint: the kappa::Lock class may be used as a base class for thread-safe locking.) It may, via the ProcessControlBlock pointer, access all Kappa library CUDA functionality.

The callback function is passed:

• void *user_data	The (void *) pointer to the user data.
• std::string name	The name of the callback.
• ProcessControlBlock *pcb	The ProcessControlBlock pointer.
• Variable *variable	The Variable argument
• ArgumentDirection direction	The direction for the Variable argument: ○ Direction_In = 0, ○ Direction_Out = 1, ○ Direction_IO = 2, ○ Direction_TexRef = 3
• bool *cancel	A pointer to a boolean used to set a CANCELED completion status: "*cancel = true;"

The callback function returns false to set a FAILED completion status or returns true and does not set the cancel pointer variable to set a FINISHED completion status.

Appendixes

Third-Party Materials

Apache

libffi

MuParser

NVIDIA CUDA Toolkit

3. TERMINATION

This LICENSE will automatically terminate if Licensee fails to comply with any of the terms and conditions hereof. In such event, Licensee must destroy all copies of the SOFTWARE and all of its component parts.

Defensive Suspension. If Licensee commences or participates in any legal proceeding against NVIDIA, then NVIDIA may, in its sole discretion, suspend or terminate all license grants and any other rights provided under this LICENSE during the pendency of such legal proceedings.

4. COPYRIGHT

All rights, title, interest and copyrights in and to the SOFTWARE (including but not limited to all images, photographs, animations, video, audio, music, text, and other information incorporated into the SOFTWARE), the accompanying printed materials, and any copies of the SOFTWARE, are owned by NVIDIA, or its suppliers. The SOFTWARE is protected by copyright laws and international treaty provisions. Accordingly, Licensee is required to treat the SOFTWARE like any other copyrighted material, except as otherwise allowed pursuant to this LICENSE and that it may make one copy of the SOFTWARE solely for backup or archive purposes.

RESTRICTED RIGHTS NOTICE. Software has been developed entirely at private expense and is commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth

in the license agreement under which Software was obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable. Contractor/manufacturer is NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA 95050

5. APPLICABLE LAW

This LICENSE shall be deemed to have been made in, and shall be construed pursuant to, the laws of the State of Delaware. The United Nations Convention on Contracts for the International Sale of Goods is specifically disclaimed.

6. DISCLAIMER OF WARRANTIES AND LIMITATION ON LIABILITY

6.1 No Warranties. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THE SOFTWARE IS PROVIDED "AS IS" AND NVIDIA AND ITS SUPPLIERS DISCLAIM ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

6.2 No Liability for Consequential Damages. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL NVIDIA OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

6.3 No Support. NVIDIA has no obligation to support or to provide any updates of the Software.

7. MISCELLANEOUS

7.1 Feedback. In the event Licensee contacts NVIDIA to request Feedback (as defined below) on how to design, implement, or optimize Licensee's product for use with the SOFTWARE, the following terms and conditions apply the Feedback:

1. Exchange of Feedback. Both parties agree that neither party has an obligation to give the other party any suggestions, comments or other feedback, whether verbally or in code form ("Feedback"), relating to (i) the SOFTWARE; (ii) Licensee's products; (iii) Licensee's use of the SOFTWARE; or (iv) optimization of Licensee's product with the SOFTWARE. In the event either party provides Feedback to the other party, the party receiving the Feedback may use and include any Feedback that the other

party voluntarily provides to improve the (i) SOFTWARE or other related NVIDIA technologies, respectively for the benefit of NVIDIA; or (ii) Licensee's product or other related Licensee technologies, respectively for the benefit of Licensee. Accordingly, if either party provides Feedback to the other party, both parties agree that the other party and its respective licensees may freely use, reproduce, license, distribute, and otherwise commercialize the Feedback in the (i) SOFTWARE or other related technologies; or (ii) Licensee's products or other related technologies, respectively, without the payment of any royalties or fees.

2. Residual Rights. Licensee agrees that NVIDIA shall be free to use any general knowledge, skills and experience, (including, but not limited to, ideas, concepts, know-how, or techniques) ("Residuals"), contained in the (i) Feedback provided by Licensee to NVIDIA; (ii) Licensee's products shared or disclosed to NVIDIA in connection with the Feedback; or (c) Licensee's confidential information voluntarily provided to NVIDIA in connection with the Feedback, which are retained in the memories of NVIDIA's employees, agents, or contractors who have had access to such (i) Feedback provided by Licensee to NVIDIA; (ii) Licensee's products; or (c) Licensee's confidential information voluntarily provided to NVIDIA, in connection with the Feedback. Subject to the terms and conditions of this Agreement, NVIDIA's employees, agents, or contractors shall not be prevented from using Residuals as part of such employee's, agent's or contractor's general knowledge, skills, experience, talent, and/or expertise. NVIDIA shall not have any obligation to limit or restrict the assignment of such employees, agents or contractors or to pay royalties for any work resulting from the use of Residuals.

3. Disclaimer of Warranty. FEEDBACK FROM EITHER PARTY IS PROVIDED FOR THE OTHER PARTY'S USE "AS IS" AND BOTH PARTIES DISCLAIM ALL WARRANTIES, EXPRESS, IMPLIED AND STATUTORY INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. BOTH PARTIES DO NOT REPRESENT OR WARRANT THAT THE FEEDBACK WILL MEET THE OTHER PARTY'S REQUIREMENTS OR THAT THE OPERATION OR IMPLEMENTATION OF THE FEEDBACK WILL BE UNINTERRUPTED OR ERROR-FREE.

4. No Liability for Consequential Damages. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL EITHER PARTY OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE FEEDBACK, EVEN IF THE OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

5. Freedom of Action. Licensee agrees that this Agreement is nonexclusive and NVIDIA may currently or in the future be developing software, other technology or confidential information internally, or receiving confidential information from other parties that maybe similar to the Feedback and Licensee's confidential information (as provided in Section 7.1.2 above), which may be provided to NVIDIA in connection with Feedback by Licensee. Accordingly, Licensee agrees that nothing in this Agreement will be construed as a representation or inference that NVIDIA will not develop, design, manufacture,

acquire, market products, or have products developed, designed, manufactured, acquired, or marketed for NVIDIA, that compete with the Licensee's products or confidential information.

6. No Implied Licenses. Under no circumstances should anything in this Agreement be construed as NVIDIA granting by implication, estoppel or otherwise, (i) a license to any NVIDIA product or technology other than the SOFTWARE; or (ii) any additional license rights for the SOFTWARE other than the licenses expressly granted in this Agreement.

7.2 If any provision of this LICENSE is inconsistent with, or cannot be fully enforced under, the law, such provision will be construed as limited to the extent necessary to be consistent with and fully enforceable under the law. This LICENSE is the final, complete and exclusive agreement between the parties relating to the subject matter hereof, and supersedes all prior or contemporaneous understandings and agreements relating to such subject matter, whether oral or written. This LICENSE may only be modified in writing signed by an authorized officer of NVIDIA. Licensee agrees that it will not ship, transfer or export the SOFTWARE into any country, or use the SOFTWARE in any manner, prohibited by the United States Bureau of Export Administration or any export laws, restrictions or regulations.

ATTACHMENT A

Redistributable Components

The following files may be redistributed with software applications developed by Licensee.

* Windows

- CUDA Runtime: cudart32_*.dll cudart64_*.dll

- CUDA FFT Library: cufft32_*.dll cufft64_*.dll

- CUDA BLAS Library: cublas32_*.dll cublas64_*.dll

* MacOS

- CUDA Runtime: libcudart.dylib

- CUDA FFT Library: libcufft.dylib

- CUDA BLAS Library: libcublas.dylib

* Linux

- CUDA Runtime: libcudart.so

- CUDA FFT Library: libcufft.so

- CUDA BLAS Library: libcublas.so

The following terms and conditions apply to Licensee's use of the components listed above ("Redistributable Components") of the SOFTWARE:

1. Customer may transfer, redistribute or sublicense, the license rights pursuant to Section 2.1.1 of this Agreement in connection with the Redistributable Components to end users of Licensee's products.

OpenSSL

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both

PCRE

End

Legal Notices

Alphabetical Index