Table of Contents

AMD GPU Compiler and Runtime Metadata and ABI Specification

Purpose

The purpose of this document is to describe both the Runtime ABI between kernels generated by the GPU compiler and runtime backends and the metadata that is passed between the compiler and the runtime. This doc will consist of three sections, the first being the Runtime ABI and the second being the compiler metadata and the final section covering an extension to the metadata that allows multiple kernels to be executed from a single IL compilation unit.

Assumptions and setup

- On both 7XX cards and evergreen series, UAV access is to UAV 1 and done on a raw buffer.
- On evergreen cards, uav id 0 is used for the arena uav. This uav should be aliased to the same piece of memory as the raw uav. Evergreen cards need to setup both the raw and arena uav's.
- When images are supported, evergreen cards will use UAV 2-8 if 7 write_only exist and 1-8 if 8 write_only images exist. In the case that 8 write_only images exist, the raw_uav which is bound to 1 is not generated.
- The compiler requires 2 constant buffers be reserved, cb0 and cb1. cb0 is reserved for internal use and cb1 is reserved for kernel specific ABI information.

Calling the compiler

The compiler now has different compiler paths for different asics and thus needs to be told what graphic cards to compile code for. The option to compile is via the -mcpu compiler option. The options are as follows:

- rv710
- rv730
- rv770 - also includes rv740, 790 and 700
- cypress - Cayman in 8XX mode
- juniper - Barts in 8XX mode
- redwood - Turks in 8XX mode
- cedar - Caicos in 8XX mode
- cayman
- barts
- turks
- caicos
- kauai - Emulator testing
- test - Test backend for features that are in developmen

The compiler also has the ability for certain abilities to passed in via the command line. However, due to limitations of llvm/tablegen, all options need to be manually specified on the command line. Therefor, in order to get a more general solution, the -mattr command line option will piggybacked on to allow the runtime to specify certain items to the compiler. The options are listed below:

- Max work group size - mwgs-#d-#1-#2-#3 where #d is the dimensionality of the max work group size, all dimensions that are not specified are 1. The values #[123] are the values for those specific dimensions. Example llc -march=amdil -mcpu=atir700 -mattr=mwgs-3-64-2-2 -regalloc=linearscan
- CAL Version - +cal=<#> where # is the CAL version that is being compiled to. Certain features will be enabled based on the CAL version. The reason is for SC versions. The current mapping is as

follows.

- SC_135 = cal=950
- SC_136 = cal=982
- SC_137 = cal=1331

Tablegen specific compiler options

Outside of the above min/max work group sizes, there are other options that can be specified by the compiler. These turn on specific features for the backend that are not on by default. They represent test features that are not fully implemented on both Apple/PC platforms or features specific to a card that require a specific version of another library to be enabled. The features are as follows, which can be found by running llc -march=amdil -mcpu=help < /dev/null

- [+|-]fp64 - Turn on/off double support.
- [+|-]byte_addressable_store - Enable/Disable byte addressable store code generation. This is only valid on Evergreen family or later and after CAL version 592.
- [+|-]barrier_detect - Turn on/off barrier detect mode.
- [+|-]images - Enable/Disable the image extension for OpenCL 1.0. This is only valid on Evergreen family or later and after CAL version 592.
- [+|-]multi_uav - Enable/Disable zero copy code generation on the GPU. This is only valid on Evergreen family of chips or later.
- [+|-]macrodb - Enable/Disable embedding of macros into the IL during codegen instead of waiting for runtime modifications.
- [+|-]noalias - Turn on/off backend assumptions on pointer aliasing.
- [+|-]noinline - Turn on/off backend inlining of functions

Runtime ABI

The runtime ABI defines the expected location of various values and how the kernel parameters are passed via the runtime to the kernel via a constant buffer. There are current 6 reserved locations and then N-6 locations are dynamically allocated by the backend compiler. There are also two function calls that need to be handled correctly, one is clEnqueueNDRangeKernel and the other is clEnqueueTask. The following tables hold the information for these two function calls. All values are specified as 32bit integers unless specified differently. Reserved Locations

Bytes	CB Loc	clEnqueueNDRangeKernel	clEnqueueTask
0-3	cb0[0].x	global_work_size[0]	1
4-7	cb0[0].y	global_work_size[1]	1
8-11	cb0[0].z	global_work_size[2]	1
12-15	cb0[0].w	work_dim	0
16-19	cb0[1].x	local_work_size[0]	1
20-23	cb0[1].y	local_work_size[1]	1
24-27	cb0[1].z	local_work_size[2]	1
28-31	cb0[1].w	0	
32-35	cb0[2].x	global_work_size[0]/local_work_size[0]	1
36-39	cb0[2].y	global_work_size[1]/local_work_size[1]	1
40-43	cb0[2].z	global_work_size[2]/local_work_size[2]	1
44-47	cb0[2].w	0	
48-51	cb0[3].x	Offset to private memory ring(0 if private memory is not emulated)	
52-55	cb0[3].y	Private memory allocated per thread	
56-59	cb0[3].z	0	

60-63	cb0[3].w	0
64-67	cb0[4].x	Offset to local memory ring(0 if local memory is not emulated)
68-71	cb0[4].y	Local memory allocated per group
72-75	cb0[4].z	0
76-79	cb0[4].w	pointer to location in global buffer that math library tables start
80-83	cb0[5].x	0.0 as IEEE-32bit float - required for math library
84-87	cb0[5].y	0.5 as IEEE-32bit float - required for math library
88-91	cb0[5].z	1.0 as IEEE-32bit float- required for math library
92-95	cb0[5].w	2.0 as IEEE-32bit float- required for math library
96-99	cb0[6].x	Global Offset for the x dimension of the thread spawn
100-103	cb0[6].y	Global Offset for the y dimension of the thread spawn
104-107	cb0[6].z	Global Offset for the z dimension of the thread spawn
108-111	cb0[6].w	Global single dimension flat offset, x * y * z
112-115	cb0[7].x	Group Offset for the x dimension of the thread spawn
116-119	cb0[7].y	Group Offset for the y dimension of the thread spawn
120-123	cb0[7].z	Group Offset for the z dimension of the thread spawn
124-127	cb0[7].w	Group single dimension flat offset, x * y * z
128-131	cb0[8].x	Offset in the global buffer to where data segment exists
132-135	cb0[8].y	Offset into buffer for printf support
136-139	cb0[8].z	Size of the printf buffer

The private memory allocated per thread and the local memory allocated per thread is passed to the runtime via the compiler metadata. The private memory is fully specified by the compiler and the local memory is a summation of what is required by the compiler and what is specified at runtime. The values passed as offsets and per thread/group allocation is all that is required to calculate safe address locations. The formulas to calculate are as follows with ptr_offset being defined in the metadata below:

```
Private offset: (vAbsTid & cb0[3].z) * cb0[3].y + cb0[3].x + ptr_offset
Local offset: (vThreadGrpIdFlat & cb0[4].z)* cb0[4].y + cb0[4].x + ptr_offset
```

Hardware Local Memory Usage

Starting with the evergreen family of chips, the compiler will generate hardware local memory operations instead of using the emulated local memory in the global buffer as was done on the HD4XXX . In order to do this correctly, local memory buffers must have their offsets handled differently. The correct way to implement this is that the offsets do not begin at 0, but begin after all the local memory space that is reported by the compiler to the runtime. For example, if a kernel has a local kernel pointer and a local memory space array, the compiler will report back to the runtime the size of the local memory space array, but not the size of the local kernel pointer since it is not known at runtime. The runtime will then place in the constant buffer the local array offset for the kernel pointer. The offset must start after the space allocated for the local memory space array. This applies also to the constant memory usage and scratch buffer where kernel parameter offsets go after the value specified in the kernel.

```
Example:
__kernel void simple_rw_mix_local(__global int2* a, int offset, __local int* d)
{
   size_t lidx = get_local_id(0);
   size_t gidx = get_global_id(0);
   __local int e[64];
   d[lidx] = gidx;
   e[lidx] = lidx;
   barrier(CLK_LOCAL_MEM_FENCE);
   a[gidx] = (int2)(d[offset], e[offset]);
}
```

Data Segment

OpenCL requires that compiler stack support constant address space pointers that are allocated globally and have pre-defined values. In order to fulfill this compiler, the compiler will generate a data segment at the end of the compilation unit if these values exist. The data segment will consist of multiple data vectors each with a varying number of data elements. The data segment itself will start with the ;#DATASTART[:CBID]:size and end with the ;#DATAEND[:CBID] tokens and each line will begin with ;# followed by the type, beginning offset, and then the size, seperated by ':' tokens. The CBID token is the constant buffer that the data is expected to reside in, with data segment with no CBID being represented as global emulation segment. Each constant buffer will have it's own data segment. The size of the whole data segment is appended to #DATASTART but delimited with a ':' token. There will then be a ':'-delimited list of values that are specified in the kernel and passed to the runtime. The data segment will be the first part of the kernel and will come before all IL instructions. Floating point values will be represented as 32 or 64bit integer hex values so that there is no error or loss of precision in the conversions. The data is allocated by the runtime in the global buffer and the offset is provided in cb0[8].x as specified in the table above. Each data segment is allocated at the offset specified by the compiler and the alignment of the data type is preserved. In OpenCL 1.1, vec3 types are introduced and they must be aligned to the 4 byte boundary. In the case of vec3, the data will be padded with zero's to align to the correct boundary.

```
Example 1:
__constant int array[4] = {0, 1, 2, 3};
Generated Data Segment:
;#DATASTART:16 <-- no ID, so is emulated global buffer
;#i32:0:4:0:1:2:3
;#DATAEND:0

Example 2:
__constant int array[3] = {20, 9123, 124};
__constant float farray[1023] = { 91, ..., 1124};

;#DATASTART:2:1040 <-- this data goes in hardware constant buffer 2
;#i32:0:3:20:9123:124
;#float:16:1023:42B60000:...:448C8000
;#DATAEND:2

Example 3:
OpenCL 1.1 and higher
__constant int3 array[2] = {(int3)(1, 2, 3), (int3)(4, 5, 6)};
;#DATASTART:32
;#v3i32:0:8:1:2:3:0:4:5:6:0
;#DATAEND:0
```

Debug Information

All debug information will be encoded in the ;DEBUGSTART/;DEBUGEND tokens. Anything between these tokens need to be stripped and should not be sent to SC.

Reserved Memory Location

Due to how SC handles kernels with no memory writes, the first 4 bytes of the private arena need to be reserved for the compiler. The compiler inserts a write of garbage data to this location when the kernel has no memory writes keeping SC from optimizing the whole kernel away. This is a workaround and not a final solution as the runtime should be able to handle an empty program. This feature will be conveyed to the runtime with an opcode in the memory section below.

Dynamic Locations

Dynamic locations are data locations that are defined at compile time and passed to the runtime from the compiler. The compiler then sets up the copies between the constant buffers and the locations where the generated code expects them to reside. There are six cases that need to be handled for the dynamic locations. The four cases are pointer arguments, aggregate arguments, scalar arguments, 32,64bit vector arguments that are less than or equal to 128bits wide, 1,8,16bit vector arguments that contain at most 4 components, 32,64bit vector arguments that are greater than 128bits wide and 1,8,16 bit vector arguments that contain more than 4 components. The location where the argument will reside in the constant buffer is specified by the compiler in the metadata explained below. How the compiler decides and sets up those locations is specified in the five sections below. All kernel arguments are 128bit, or 16byte aligned, and start at byte 96.

Scalar Arguments

The scalar arguments are passed to the kernel at the offset specified by the metadata. The alignment is 128bits and the values are passed unmodified. Sub-32bit data types are not extended to fit into a 32bit type. For 64bit data types, the lower 32 bits are stored in the first 4 bytes, or the x component, and the upper 32 bits are stored in the second 4 bytes, or in the y component. These take up a single vector location in constant buffer 0. The compiler will generate move instructions to move them to the correct registers. The value passed in this location is the equivalents of a pass by value in C. Boolean scalars are equivalent to a 32bit integer with the upper 31bits masked out.

Pointer Arguments

Pointer arguments are special in that they do not pass in any explicit data or a pointer in the C sense. A pointer argument will have in the first 32 bits the offset into its respective memory space that the memory is allocated to. This means in simpler terms that the value passed in the pointer argument is the address of the first location where data can be found. The alignment constrains are the same as Scalar arguments, but the address must be aligned on the 128bit boundary.

Aggregate Arguments

Aggregate arguments such as unions and structs are passed similar to their scalar types. For unions they are passed as if they were equivalent to their largest type and follow the rules that involve that type. For structures, they are aligned to the 128bit boundary and take up (sizeof(struct) + 127) & ~127 bits of space aligned on the 128bit boundary. In byte terms, they take up the size of the struct rounded up to the next multiple of 16 bytes and 16 byte aligned, or (sizeof(struct) + 15) & ~15.

1, 32,64bit Vector Arguments <= 128bits wide

The vector arguments that fit naturally into the constant buffer and are not larger than 128bits wide have the same rules applied as the scalar arguments. Each component of the vector must align on the 32 or 64bit boundary depending on the bit width of the base type of the vector. These arguments take up a single vector location in constant buffer 0. The compiler will generate move instructions to move them to the correct registers. The value passed in this location is the equivalents of a pass by value in C. Boolean vectors are treated equivalent to a 32bit vector that has its upper 31 bits masked out.

1, 32, 64bit Vector Arguments > 128bits wide

The 32 and 64bit vector arguments which are larger than 128bits wide must be treated differently. The compiler in this case will generate multiple move instructions and move the 128 bit types into sequential registers. These arguments take up 1 vector location in constant buffer 0 for each 128bits they require. For example, a float8 requires 2 vector locations, but a double8 requires 4 vector locations. These arguments must

be placed in the locations specified in the metadata and each component must be aligned and packed respective of its type. The value passed in this location is the equivalents of a pass by value in C. Boolean vectors are treated equivalent to a 32bit vector that has its upper 31 bits masked out.

8, 16bit Vector Arguments <= 4 components

Vectors arguments that fall into this category need to be handled by the compiler different since a vector in the hardware is only 4 components. All vector arguments in this category are expected to be aligned at their types. For example a char2 should be packed into the first 16 bits of a 32bit component. Each of these arguments only take up one vector slot and are unpacked into a 4 component vector.

8, 16bit Vector Arguments > 4 components

Vector arguments that fall into this category have their data packed and must be unpacked into multiple registers. The data should be packed but every 4 components align on the 128bit boundary. This allows for the compiler to easily copy the data without doing lots of computation per thread.

Image Metadata

Image arguments are treated differently than normal pass by value arguments. They take up two sequential slots or 32 bytes of data. The data, which is in 32bit floats, is setup in the following manner:

bytes	7XX	8XX	notes
0-3	width	width	int format
4-7	height	height	int format
8-11	depth	depth	(for 2D image, this must be 1)
12-15	offset	data_type	
16-19	data_type	1.0f/width	8XX data is in FP format
20-23	channel_order	1.0f/height	8XX data is in FP format
24-27	row_pitch	1.0f/depth	8XX data is in FP format
28-31	slice_pitch	channel_order	

Sampler Metadata

The sampler metadata is passed as a 32bit integer with the following information encoded in it. The X component is filled with values in the following table. TODO: fill out table

Compiler Metadata

The compiler metadata is created by the compiler to pass data back to the runtime so that the runtime can correctly set data as the generated kernel expects it. There metadata creates metadata for many things that need to be passed between the kernel and runtime. This includes information about arguments, memory space requirements for private/local memory, information on the group size and how memory pointers are to be used.

String representation

All argument sequences in the string representation start with ;ARGSTART:func_name and end with ;ARGEND:func_name. All arguments for the specific function are inside of this block and are listed once per line starting with a ‘;’ followed by a token specifying the section they belong to.

Metadata Version Number

In order to correctly version the metadata a trio of numbers is being added to the spec. These numbers correspond to major version, minor version and revision number. The major version number only increases when the metadata breaks backward compatibility with the previous metadata versions. The metadata parser must accept all minor versions previous to the current version as all minor version changes should be backward compatible. The revision number shall be auto-incremented via some mechanism to be determined based on the current infrastructure and is only used in helping determine the exact version of the compiler being used. This information can be ignored by the runtime.

;version:Major:Minor:Revision

1. Major: Version of the metadata, incompatible with previous major versions
2. Minor: Version of the metadata that is compatible with previous minor changes
3. Revision: unique number specifying metadata revision that matches up with twiki document

Device name

In order to help with debugging, the name of the device the IL was compiled for is embedded in the metadata section for each kernel.

;device:Name

1. Name: Name of the device the IL was compiled for.

Error Token

A token that specifies that an error has occured and that the runtime should return failure on compilation. This can occur multiple times per kernel as multiple errors might occur. They can occur in the kernel wrapper or in the function. How these are reported back to the user is implementation dependent. The error message will be no more than 32 characters long and be prefixed by a 4 character condition code.

;error:msg

- msg - The reason for the error

Warning Token

A token that specifies that a warning has occured and that the runtime should return the message in the compilation log. This can occur multiple times per kernel as multiple warnings might occur. They can occur in the kernel wrapper or in the function. How these are reported back to the user is implementation dependent. The warning message will be no more than 32 characters long and be prefixed by a 4 character condition code.

;warning:msg

- msg - The reason for the warning

Local/Private Memory

The amount of local memory or private memory that is required for a specific kernel is passed back to the runtime with the following pattern:

;memory:AddrSpace:MemSize

1. Address space: The address space to add this memory to, either ‘local’, ‘hwlocal’, ‘private’ or ‘hwprivate’
2. Memory size: The amount of memory to add to the total for this address space

This argument is not required to be unique and the real value that is required is the sum of all the types for that specific kernel and all the functions that it calls. If hardware memory is being used instead of emulated in global, then the kernel parameter offsets must be shifted by the total amount specified from this value. The difference between hw and non hw memory spaces is that the non-hw can use the Arena for sub-32bit writes.

Reserved Memory Write

In order to pass conformance, some tests need a memory write to be inserted into the compiler. This token will only exist if no user-specified memory write exists. The token is as follows:

;memory:compilerwrite

Data Segment Memory

This flag lets the runtime know if the data segment is required to be allocated for this kernel. This flag only shows up on the condition that a data segment appears and that this specific kernel requires the usage of the data segment. See the data segment section for more information on the setup. The runtime also must keep track of data segment tokens of subsequent function calls. This token is provided as an optimization to let the runtime know when the data needs to be copied and when it is not a requirement. All kernels in a single compilation unit use the same data segment. The token is as follows:

;memory:datareqd

Unique Kernel ID

This token specifies the mapping between a kernel and its unique ID. The format is in the following pattern:

;uniqueid:ID

1. ID: An integer value that is uniquely mapped to this kernel

This value is unique per kernel and cannot appear more than once.

Sampler Arguments

Samplers can be defined in the kernel or as a kernel argument. Samplers that are defined as kernel arguments are defined in the Pass by value section. The bitfield is defined in the Sampler Metadata section. Kernel Sampler arguments must have their data passed back to the kernel via the constant buffer specified in the pass by value argument location. The pattern is as follows:

;sampler:ArgName:id:location:value

1. Argument Name: The name of the argument that image this maps to.
2. ID: The id of this sampler
3. Location: 1 for kernel defined, 0 for kernel argument
4. Value: The bitfield value of this sampler as a 32bit integer, 0 if a kernel argument

Image Kernel Arguments

Image arguments have 7 tokens that specify information around that specific argument. The data specified in the Image Metadata section shall be passed in the Image Metadata section above. The pattern is as follows:

;image:ArgName:dimension:type:id:ConstNum:ConstOffset

1. Argument Name: The name of the argument that image this maps to.
2. Dimension: 2D or 3D for the dimensionality
3. Type: RO for read-only, WO for write-only, RW for read-write images
4. ID: The uav/sampler ID that corresponds to this image
5. Constant buffer number: The constant buffer that this data is expected to be mapped to
6. Constant offset: The offset in bytes that this data is expected to be

Counter kernel arguments

Counter arguments are special pointers that are only used if the cl_amd_atomic_counter extension is enabled. These arguments have 6 tokens that specify information for the specific argument. The pattern for this argument is as follows:

;counter:ArgName:EleSize:id:ConstNum:ConstOffset

1. Argument Name: The name of the argument that the counter maps to.
2. Element Size: Field that specifies either a 32bit or 64bit counter in bits.
3. ID: The ID into the GDS that this counter is expected to be bound to.
4. Constant buffer number: The constant buffer that this data is expected to be mapped to.
5. Constant offset: The offset in bytes that this data is expected to be located at.

Pass by value arguments

Pass by value arguments have 6 tokens that specify information around that specific argument. The pattern is as follows:

;value:ArgName:Datatype:Size:ConstNum:ConstOffset

1. Argument Name: The name of the argument that this maps to.
2. Data type: either “i1”, “i8”, “i16”, “i32”, “i64”, “float”, “double”, “struct”, “union”, “event” , “opaque”
3. Num Element: either 1, 2, 3(OpenCl 1.1 only), 4, 8 or 16, where all scalars, samplers, events and opaque are 1, vectors are 2-16, structs hold the total size of the struct in bytes, and unions hold the size of the largest element in bytes
4. Constant buffer number: The constant buffer that this data is expected to be mapped to
5. Constant offset: The offset in bytes that this data is expected to be

Pass by pointer arguments

Pass by pointer arguments are pointers in memory that require some data to be passed down to the kernel such has base offset. This metadata string has 8 tokens and follows the following pattern:

;pointer:ArgName:DataType:NumEle:ConstNum:ConstOffset:MemType:BufNum:DataTypeAlignment

1. Argument Name: The name of the pointer argument
2. Data type: either “i1”, “i8”, “i16”, “i32”, “i64”, “float” or “double”.

3. Num Element: This will be set to 1 as there cannot be a pointer vector/array of pointers
4. Constant buffer number: The constant buffer that this data is expected to be mapped to
5. Constant offset: The offset in bytes that this data is expected to be written to
6. Memory Type: One of the following memory types that specify how this memory should be allocated
    a. g - this pointer is expected to point to global memory
    b. p - this pointer is expected to use emulated private memory
    c. l - this pointer is expected to use emulated local memory
    d. uav - this pointer is expected to use a uniform access vector
    e. c - this pointer is expected to use emulated constant buffer
    f. r - this pointer is expected to use emulated GDS memory(If the extension is supported)
    g. hl - this pointer is expected to use hardware local memory
    h. hp - this pointer is expected to use hardware private memory
    i. hc - this pointer is expected to use hardware constant buffer
    j. hr - this pointer is expected to use hardware GDS memory(If the extension is supported)
7. Buffer Number: The data is expected to be the the nth numbered buffer of its respective type
8. Data Type Alignment: The alignment of the data that this pointer points to in bytes

Example: • __global int* result ;ARG:main:result:pointer:1:0:96:g:0:4 • float x ;ARG:main:x:float:1:0:96

Raw UAV ID number

This token specifies what ID the raw uav should be declared at in the case where a UAV is used, but no kernel arguments use a UAV. Examples of this include, printf, global constants or compiler generated scratch to Emulated Private. If uavid is equal to 8, then no raw uav is being used and the arena uav only is being used. This token is only valid for 7XX/EG/NI devices.

;uavid:

1. 1. a. ID - ID of raw UAV.

Printf information

In order to support the printf function for debugging. The GPU needs specific metadata to handle how printf should be handled. What printf does is pass the format string in the metadata to the runtime and gives a specific ID that the metadata corresponds to. The arguments are then written to a buffer location that is specified in CB0[8].y as specified in the Runtime ABI section. Any printf writes exceeding the size specified in CB0[8].z are silently dropped. The printf token will follow the following format:

;printf_fmt:ID:numArgs:ArgNSize*:strLen:fmtStr

- ID - The globally unique ID that this string corresponds to.
- numArgs - The number of arguments that the format string requires.
- argNSize - For each argument, the size of the argument data type.
- strLen - The length of the format string.
- fmtStr - The format string for the printf function.

```
Examples:
printf("!---initizeWSDeque---!\n");
;printf_fmt:0:0:23:!---initizeWSDeque---!\n;
printf("Bottom: %d Top: %d InitialSize: %d\n", 1, 2, 3);
;printf_fmt:4:3:4:4:4:35:Bottom: %d Top: %d InitialSize: %d\n;
```

Multiple Kernel Compilation

In order to conform to the OpenCL spec, multiple kernels need to be specified in a single file and the compiler must handle them without error in compilation. The main drawback to this is that IL and the SC compiler have no notion of multiple entry points in a single compilation unit. This section will describe the approach taken by the OpenCL GPU backend that to create a solution for this problem. This approach involves modification of the compilation process and also introducing new metadata and dynamic manipulation of the generated code by the runtime. Compiler Modifications In a fully functional IL kernel, all the declarations and functions are explicitly defined and all functions calls should be decided at code generation time. However, in order to allow for multiple kernels, or 'main' functions as IL defines them, a pseudo instruction will be created. This pseudo instruction will start with a ';' and consist of 10 '$' symbols, so the pseudo-instruction will be ;$$$$$$$$$$. This will be embedded in the prolog code that the compiler will generate which will contain the following items:

- IL setup code and declarations
- a superset list of all literals used in all kernels
- Code to calculate intrinsic work-item functions
- the aforementioned pseudo-instruction
- endmain

This will generate everything that will from now on be called the 'main' function. This does not contain any code that is used in an OpenCL function and thus should be static for all OpenCL kernels. For each kernel function, the top of each function will be a copy of arguments from the specified constant buffer to the correct registers where llvm expects them to be. At the end of each kernel function will be metadata that specifies the following information.

- The metadata for this function
- A list of all function ID's that are called by the current function
- A list of all intrinsics that need to be linked in when this kernel is called

Multi-Kernel Metadata

There is new metadata that must be used for multi-kernel compilation units. The first new piece of metadata is the call graph for the current function; the second piece is the intrinsic library. The reason that they are separate is that the functions are required to be in the original IL string, whereas the intrinsics should be linked in from a separate library. Function Metadata The function metadata provides a call graph for this specific function to the runtime without having to reparse the data. This allows for the runtime to quickly prune functions that are not used from being passed to the lower level compiler. There can be multiple instances of this metadata.

Format

;function:NumFuncs:UID0:UID1:…:UIDN-1

1. Num Funcs - Number of functions included in this instance of the metadata
2. UID0-UIDN-1: The unique identifiers of the functions that are included

Intrinsic Metadata

The intrinsic metadata is exactly the same as the function metadata, except that the functions are not expected to be included in the original IL string and must be retrieved from another source base.

Format

;intrinsic:NumIntrs:UID0:UID1:…:UIDN-1

1. Num Intrs - Number of intrinsics included in this instance of the metadata
2. UID0-UIDN-1: The unique identifiers of the intrinsics that should be included

Per Kernel Local Group Size

In opencl, a programmer is able to specify per kernel local group size. This information needs to be conveyed to the runtime and is done via the LWS field of the metadata. The metadata specifies non-standard group information, where standard group information is specified via 64, 1, 1.

Format: ;cws:First:Second:Third

1. First : size of the first dimension
2. Second : size of the second dimension
3. Third : size of the third dimension

Compiler Specified Group Size

In some cases, based on information available to the compiler, the group size must be restricted for a specific kernel. The compiler will pass this information in one of two fields. If the compiler determines the exact size that is required, then the lws field will be specified, otherwise the limitgroupsize field will be specified. The formats are below.

Format

;lws:size

1. Size : maximum size of the group when W * H * D is calculated

Format: ;limitgroupsize Limit the groupsize as determined by the runtime an ideal size for the specific chip.

Runtime Source Modification

The limitations of IL cause the runtime to finish the last step of the compilation process. This is the linking process. The runtime must link together, based on the list of functions and intrinsics that are required for each function that is in use in a kernel. Based on the super-set of function ID’s and instrinsics, the list of functions need to be appended to the end of the IL. After this step is done, the runtime must then replace the token “;$$$$$$$$$$” with the IL “call #\n” where # is to be replaced with the unique identifier of the kernel being called.

An optimization step is to split all functions into an array of strings and then only append the functions that are required by the current kernel. This will massively improve the speed at which SC compiles down to ISA from IL.

-- MicahVillmow - 04 Aug 2009