Compilation overview

For GPU support, the compiler wrapper is used, it's built by the project cmake. The wrapper is called instead of the host compiler; it passes normal files through, but handles .CXX files differently.

In emulation mode, the wrapper only sets `HOSTCODE=1` and `DEVCODE=1` and passes the files to the host compiler.

In GPU mode, the wrapper extracts the GPU part (`#if DEVCODE`), compiles it with GPU compiler, translates the resulting GPU binary to C array of bytes, merges it with CPU code (`#if HOSTCODE`) and finally compiles the result with the host compiler.

Basic types

Integer types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`.
Float types: `float16`, `float32`, `float64`.

For float types, NAN is supported. To check a number validity, use `def(X)` function.

To convert a number, use functions "`convertNearest`", "`convertDown`", "`convertUp`" like this: `convertNearest<int32>(3.1415f)`.

Type `Space` is a signed integer type to hold the maximum size or index of any image/array, currently it's signed 32-bit integer (can be less than pointer size).

Headers: "`intBase.h`" "`floatBase.h`" "`intType.h`" "`floatType.h`" "`space.h`"

Vector types

There are 2-component and 4-component vector types:

```
float32_x2 vec = make_float32_x2(0, 0);

float32 sum = vec.x + vec.y;

uint32_x4 color = make_uint32_x4(0, 0, 0, 0);

int32 sum = color.x + color.y + color.z + color.w;
```

These vector types are similar to those used on GPUs in CUDA and OpenCL.

Per-element operations are supported, binary arithmetic `+`, `-`, `*`, `/`, comparisons, etc.
Conversions are supported, you can specify scalar target type:

```
float32_x2 floatPos = 2.f * inititalPos + 1.f;

int32_x2 intPos = convertDown<int32>(floatPos);
```

Unary and binary functions are supported, such as `absv`, `minv`/`maxv`/`clampRange`/`clampMin`/`clampMax` and so on.

Vector comparisons:

```
int32_x4 value = pos;

bool_x4 result = (value >= 0) && (value <= maxValue);
```

To accumulate AND/OR of a vector's components, use functions `allv`/`anyv`:

```
if (anyv(pos < 0)) error();
```

Some functions/macros imply `allv`:

```
if_not (pos >= 0) error(); // uses “allv” before decision

require(pos >= 0); // return false if any of the components is negative
```

Headers: “vectorBase.h” “vectorTypes.h” “vectorOperations.h”

Point<T> type

`Point<T>` is a vector type with two fields, `X` and `Y`.

It supports the rich set of arithmetic and conditional operations, conversion and conditional operations.

Examples of usage:

```
// Default constructor: uninitialized
Point<int> uninitializedValue;
uninitializedValue.X = 0;

// Create a point by components. The scalar version creates a point
// with both components having the same value.
Point<int> A = point(16, 32);
Point<int> B = point(2); // B = {2, 2}

// Convert
Point<float> c0 = convertNearest<float>(A);
Point<float> c1 = convertNearest<Point<float>>(B);
Point<float> c2 = convertNearest<Point<float>>(1);

// Convert with success flag.
Point<int> intResult;
Point<bool> success = convertNearest(c2, intResult);

// Generate NAN
// Is it definite?
```

```
Point<float> pointNan = point(nanOf<float>());
Point<bool> pointDef = def(pointNan);

// Arithmetic operations.
Point<int> testUnary = -A;
Point<int> C = A + B;
Point<int> D = 2 * A;

// Assignment arithmetic operations.
A += 1;
C &= D;

// Vector comparisons.
Point<bool> Q = (A == B);
Point<bool> R = (1 <= B);

// Vector bool operations.
Point<bool> Z = Q && R;
Point<bool> W = Q || R;
Point<bool> V = !Z;

// Reduction of Point<bool> to bool.
require(allv(A >= 0));
if (anyv(A < 0)) return false;

// Min, max, clamp functions.
Point<int> t1 = minv(A, B);
Point<int> t2 = maxv(A, B);
```

```
Point<int> t3 = clampMin(A, 0);
Point<int> t4 = clampMax(A, 10);
Point<int> t5 = clampRange(A, 0, 10);
```

More real example:

```
Point<Space> srcIdx = point(320, 240);
Point<float32> srcPos = convertFloat32(srcIdx) + 0.5f;
Point<float32> dstPos = factor * srcPos;
Point<Space> nearestIdx = convertNearest<Space>(dstPos - 0.5f);
Point<bool> isValid = (nearestIdx >= 0) && (nearestIdx < size);
clampedIdx = clamp(nearestIdx, point(0), size-1);
if (allv(isValid)) {...}
```

Headers: “pointBase.h” “point.h”

Point3D<T> and Point4D<T>

Totally the same as Point<T> but with {X, Y, Z} and {X, Y, Z, W} components.

Basic control macros

`if_not (cond)`	Equivalent to `if (!(cond))`.
`while_not (cond)`	Equivalent to `while (!(cond))`.
`require(cond)`	Returns false if the condition is not true (in non-exception compilation mode).
`check_flag(cond, ok)`	If the condition is not true, sets the flag to false.

For a vector bool argument, all these macros take a logical AND of components: function `allv(cond)`.

`COMPILE_ASSERT(X)` works at compile time: if the condition is not satisfied, the compilation fails.

Headers: "`compileTools.h`"

Kits

A kit is a structure for passing parameters that are rarely changed between functions.

A kit contains several parameters. When you define a kit, you enumerate parameter types and names:

```
KIT_CREATE2(MyParams, int, A, char*, B);
```

A kit defines a constructor by kit parameters (in the same order):

```
void myFunc()
{
    MyParams params(2, "test");
}
```

A kit can be implicitly converted to any unrelated kit if required fields are convertible:

```
KIT_CREATE2(MyKit, int, number, char*, message);
KIT_CREATE3(OtherUnrelatedKit, char*, message, int, number, float, alpha);

void myFunc(const OtherUnrelatedKit& otherKit)
{
    MyKit kit = otherKit; // Works despite the kit types are unrelated
}
```

A number of kits can be combined into a single kit:

```
KIT_CREATE1(RateKit, int, rate);
KIT_CREATE1(AlphaKit, float, alpha);
```

```
KIT_COMBINE2(UnitedKit, RateKit, AlphaKit);
using AnotherUnitedKit = KitCombine<RateKit, AlphaKit, OtherKit>; // modern way

void myFunc(const UnitedKit& kit)
{
    int r = kit.rate;
    float a = kit.alpha;

    AlphaKit alphaKit = kit;
    RateKit rateKit = kit;

    UnitedKit newKit = kitCombine(alphaKit, rateKit); // combine at run-time
}
```

Often a kit is used to pass interfaces, for example:

```
struct MyConsole
{
    virtual void print(char* msg) =0;
};

KIT_CREATE1(MyConsoleKit, MyConsole&, myConsole);

void myFunc(const MyConsoleKit& kit)
{
    kit.myConsole.print("Hello");
}
```

Headers: “kit.h”

Error Handling

Currently, there are two compilation modes: via C++ exceptions and via returning bool.
Header "`errorHandling.h`" chooses the mode by `HEXLIB_ERROR_HANDLING` value.

There is a unified interface for both modes:

	Error code mode	Exceptions mode
`stdbool`	• In Release: `bool` • In Debug: A class with `[[nodiscard]]` attribute.	Empty class.
`returnTrue`	`return true`	`return stdbool{}`
`returnFalse`	`return false`	`throw ExceptFailure{}` `ExceptFailure` is an empty class.
`require(cond)`	`if (!(cond)) return false`	`if (!cond) throw ExceptFailure{}`
`require(stdbool)`	As usual.	Nothing.
`errorBlock(action)`	Pass bool.	Suppresses all exceptions and converts to bool.

Standard function macros

In the project, almost all big functions are in ‘standard’ form:

```
stdbool myFunc(int myParam, stdPars(MyKit))
{
    returnTrue;
}
```

The `stdPars(MyKit)` macro is used to pass a function kit (a set of instruments) and callstack/profiler support parameters. The kit parameter is declared with the name ‘`kit`’.

Here is how a function usually calls another standard function:

```
stdbool otherFunction(stdPars(OtherKit))
{
    require(myFunc(3, stdPass));
    returnTrue;
}
```

Here the other function uses macro `stdPass` to pass a kit with callstack/profiler support. A `kit` which is available at the call point should be convertible to the target kit. The calling function uses `require` to check the returned success flag.

Headers: “`stdFunc.h`”

Error log usage

To check a condition silently, use `require(cond)`, if the condition is failed, the function fails without messages.

If an error condition is not expected to happen normally, use `REQUIRE(cond)`. On failure, it also writes a message to the error log with a call stack. Example:

```
stdbool MyClass::myFunc(stdPars(ErrorLogKit))
{
    require(isActive); // fails silently
    REQUIRE(allocatedSize >= 0); // on failure, displays a message

    returnTrue;
}
```

To use `REQUIRE` macro, the function kit should contain `ErrorLogKit`.

Both "`require`" and "`REQUIRE`" macros use AND-reduction for a vector bool argument.

If you don't need to return from the function, use `CHECK` macro:

```
if (CHECK(pos >= 0))
    {doSomeWork();}
```

Here if the condition is failed, `CHECK` macro writes a message to the error log and returns `false`, thus skipping "`if`" body.

Headers: "`errorLog.h`"

Matrix<T> and Array<T>

The `Matrix<T>` class is a structure describing the memory layout of a 2D image:

`ptr`	`T*`	Pointer to (0, 0) element. Can be undefined if the matrix is empty.
`pitch`	`Space`	Pitch expressed in elements, not in bytes. The difference of pointers to (X, Y+1) and (X, Y) elements. Can be negative.
`size`	`Point<Space>`	The width and height of the matrix. Both are >= 0. If either of them is zero, the matrix is empty.

The `Array<T>` does the same but for 1D array this having only `ptr` and `size` fields.

The `Matrix<T>` and `Array<T>` classes are not memory owners, they don't do any memory allocation or deallocation.

Usage examples:

```
// Create an empty matrix.
Matrix<int> intMatrix;

// Convert a matrix to a read-only matrix.
Matrix<const int> constIntMatrix = intMatrix;
Matrix<const int> anotherConstMatrix = makeConst(intMatrix);

// Construct matrix from details: ptr, pitch and size.
Matrix<const uint8> example(srcMemPtrUnsafe, srcMemPitch, srcSizeX, srcSizeY);
```

```
// Setup matrix from details: ptr, pitch and size.
example.assign(srcMemPtrUnsafe, srcMemPitch, srcSizeX, srcSizeY);

// Make the matrix empty.
example.assignNull();

// Access matrix details (exposing matrix is another way):
REQUIRE(example.memPtr() != 0);
REQUIRE(example.memPitch() != 0);
REQUIRE(example.sizeX() != 0);
REQUIRE(example.sizeY() != 0);

// Expose a matrix to detail variables:
MATRIX_EXPOSE(example);
REQUIRE(exampleMemPtr != 0);
REQUIRE(exampleMemPitch != 0);
REQUIRE(exampleSizeX != 0);
REQUIRE(exampleSizeY != 0);

// Access some element in an exposed matrix.
// The macro uses multiplication. No X/Y range checking performed!
int value = *MATRIX_POINTER(example, 0, 0);
value = MATRIX_ELEMENT(example, 0, 0);

// Example element loop (not optimized):
uint32 sum = 0;
```

```
for (Space Y = 0; Y < exampleSizeY; ++Y)
    for (Space X = 0; X < exampleSizeX; ++X)
        sum += exampleMemPtr[X + Y * exampleMemPitch];

// Save rectangular area [10, 30) as a new matrix using
// "subs" (submatrix by size) function. Check that no clipping occured.
Matrix<const uint8> tmp1;
REQUIRE(example.subs(point(10), point(20), tmp1));

// Save rectangular area [10, 30) as a new matrix using
// "subr" (submatrix by rect) function. Check that no clipping occured.
Matrix<const uint8> tmp2;
REQUIRE(example.subr(point(10), point(30), tmp2));

// Remove const qualifier from element (avoid using it!)
Matrix<uint8> tmp3 = recastElement(tmp2);

// Check that matrices have equal size.
REQUIRE(equalSize(example, tmp1, tmp2));
REQUIRE(equalSize(tmp1, tmp2, point(20)));

// Check that a matrix has non-zero size
REQUIRE(hasData(example));
REQUIRE(hasData(example.size()));
```

For arrays, everything is the same, but 1D, use class `Array<T>`.

Headers: “`matrix.h`” “`array.h`”

GpuMatrix<T> and GpuArray<T>

These classes are equivalent to `Matrix<T>` and `Array<T>`, but for GPU memory.

Instead of `T*` pointer, they contain a pointer of type `GpuPtr(T)`. This pointer has type `T*` if the code is compiled for GPU, but for CPU compilation, the pointer is emulated with a special class such that behavior is identical to GPU pointer.

Headers: "`gpuMatrix.h`" "`gpuArray.h`"

Memory allocation

CPU and GPU memory is allocated via custom real-time allocators.

[In GPU mode, CPU memory is allocated from special “pinned” CPU memory].

There are two types of memory: *state memory* and *temporary memory*.

- State memory allocations are kept between processing of input frames;
- Temporary memory allocations only exist during processing of a single frame.

Both CPU and GPU memories are divided into state memory and temporary memory.

For each type of memory, the project requests only one contiguous block of memory from the system, and internally distributes it in a very fast way.

State memory is allocated in “`realloc`” function, which is called only at frame size or mode change; temporary memory is allocated in “`process`” function, which is called to process a single frame.

The “`realloc`” function, as well as the “`process`” function, is always called twice: to count memory and to do real work. The mode is specified in “`kit.dataProcessing`” variable.

- On the first pass, `kit.dataProcessing` is `false`, and all memory allocations return fake zero pointers; the application does all allocations as usual, but it shouldn’t do any real data processing.
- On the second pass, `kit.dataProcessing` is `true`, all allocations return real pointers, and the application does real data processing.

Temporary memory is allocated and deallocated by “stack” principle: alloc A, alloc B, dealloc B, alloc C, dealloc C, dealloc A. So, you can free only the last allocated block. At the end of processing, all temporary memory should be deallocated.

State memory is discarded and reallocated completely on each “`realloc`”, deallocation requests are ignored.

Memory containers

To allocate memory, use `MatrixMemory<T>`, `ArrayMemory<T>`, `GpuMatrixMemory<T>`, `GpuArrayMemory<T>`.

Example:

```
Point<Space> size = point(320, 240);
GpuMatrixMemory<float32> tmpMatrix;
require(tmpMatrix.realloc(size, stdPass));
```

It will free the memory in the destructor.

For temporary memory, macro shortcuts are available:

```
GPU_MATRIX_ALLOC(tmpMatrix, float32, size);
```

There are advanced containers: `GpuLayeredMatrixMemory<T>`, `PyramidMemory<T>` and others.

Headers: “`matrixMemory.h`” “`arrayMemory.h`” “`gpuMatrixMemory.h`” “`gpuArrayMemory.h`”

GPU processing

There is a very simple to use tool, `GPU_TOOL`:

```
GPUTOOL_2D
(
    addValueToImage,
    PREP_EMPTY,

    ((const uint8, src))
    ((uint8, dst)),

    ((int, value)),

    *dst = *src + value;
)
```

The `GPUTOOL` is a macro with parameters:

- Function name.
- List of input textures (which can have different sizes).
- List of input/output images (which all should have the same size).
- List of parameters.
- Iteration body.

A list is passed in format `(A) (B) (C)` where `A B C` are elements, each element in parenthesis, without commas.

For images and parameters, each list element is a pair `(Type, name)`, thus resulting in double parenthesis, for example: `((int, A)) ((float, B))`.

Texture list elements additionally have “interpolation mode” and “border mode”.

To pass an empty list, use `PREP_EMPTY` macro.

More realistic example:

```
GPUTOOL_2D_BEG
(
    myFunction,

    // List of textures
    ((const uint8, myLuma, INTERP_LINEAR, BORDER_MIRROR))
    ((const uint8_x2, myChroma, INTERP_LINEAR, BORDER_MIRROR)),

    // List of matrices
    ((uint8_x4, resultYuv))
    ((uint8, resultGray)),

    // List of parameters
    ((float32, coeffMul))
    ((float32, coeffAdd))
)
#if DEVCODE
{
    // In textures, first sample has a coordinate of 0.5
    // So, this position hits exactly to our sample, no interpolation:
    Point<float32> pos = point(X + 0.5f, Y + 0.5f);

    // In textures, coordinates are normalized to [0, 1] range,
```

```
    // so we need to divide the position by image size.
    float32 lumaValue = tex2D(myLumaSampler, pos * myLumaTexstep);

    // Let's suppose chroma has twice less resolution (bilinear interpolation happens).
    // Note that the texture unit expands uint8 values to float range [0, 1].
    float32_x2 chromaValue = tex2D(myChromaSampler, 0.5f * pos * myChromaTexstep);

    // Assemble a value
    float32_x4 value = make_float32_x4(lumaValue, chromaValue.x, chromaValue.y, 0);

    // Scale the value using vector operations:
    value = coeffMul * value + coeffAdd;

    // Scale range [0, 1] to 0..0xFF, round to nearest integer,
    // clamp to [0, 0xFF] and store as uint8:
    storeNorm(resultGray, value.x);
    storeNorm(resultYuv, value);
}
#endif
GPUTOOL_2D_END
```

How to call this function? On CPU side, it has such prototype:

```
bool myFunction
(
    const GpuMatrix<const uint8>& myLuma,
    const GpuMatrix<const uint8_x2>& myChroma,
    const GpuMatrix<uint8_x4>& resultYuv,
    const GpuMatrix<uint8>& resultGray,
    const float32& coeffMul,
    const float32& coeffAdd,
    stdPars(GpuProcessKit)
);
```

Note that all parameters are passed as "`const T&`", the texture images and humble images are passed identically, and the function has standard parameters with `GpuProcessKit`.

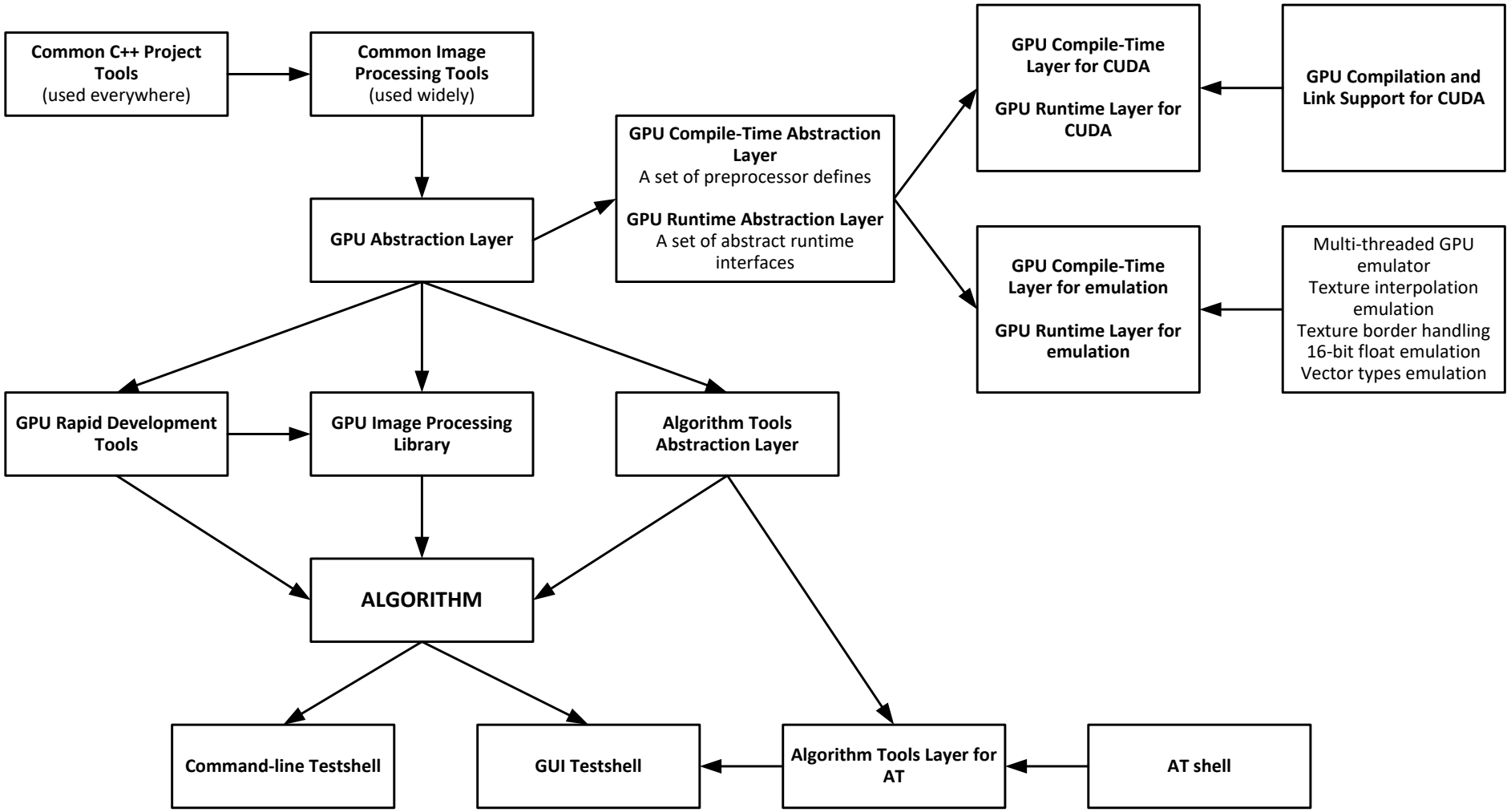
If you are familiar with GPU programming and need more advanced processing, such as caching to SRAM memory, use `GPUTOOL_2D_EX` family:

```
GPUTOOL_2D_BEG_EX(prefix, tileSize, keepAllThreads, samplerSeq, matrixSeq, paramSeq)
```

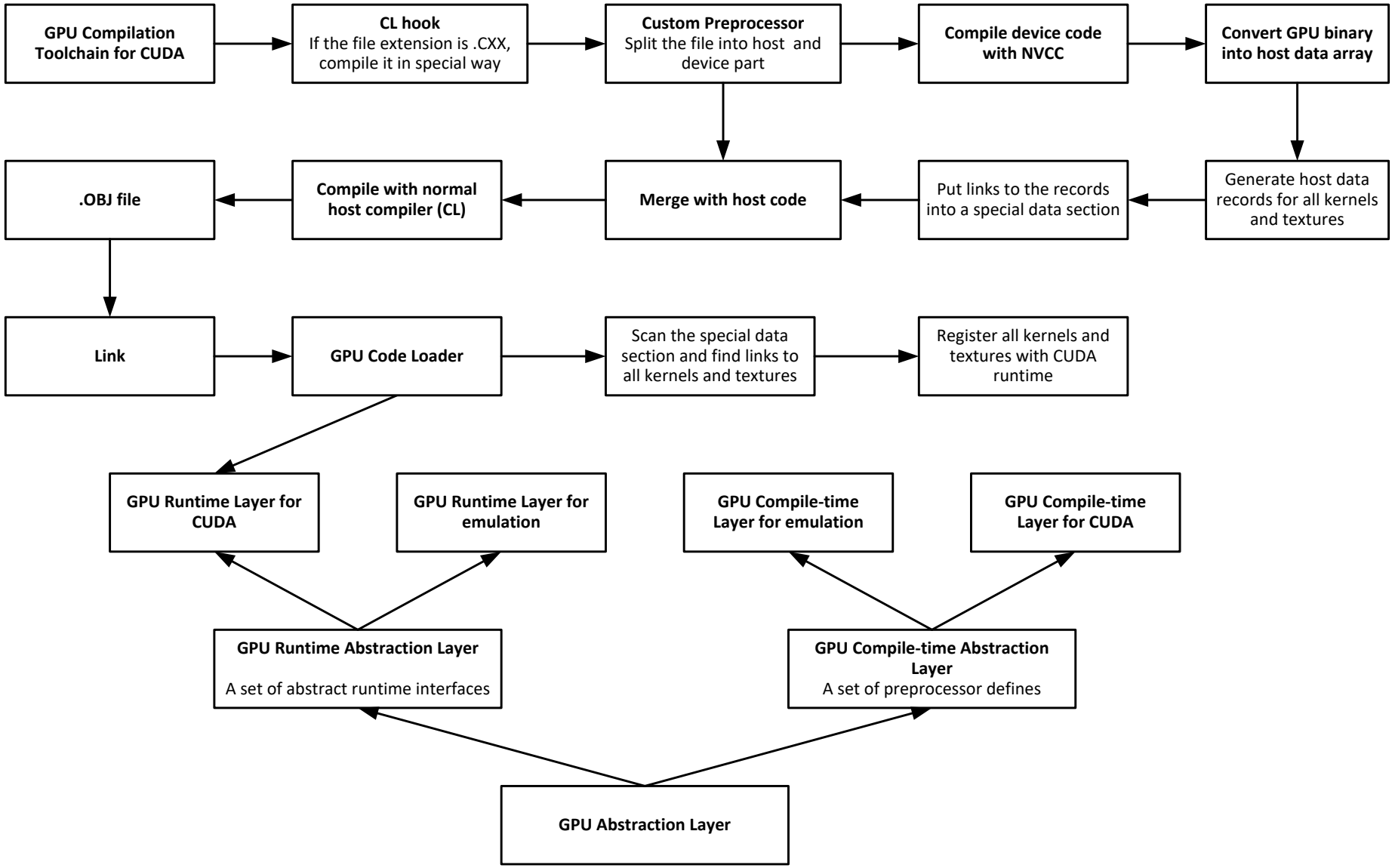
Here you additionally specify group size as a compile-time constant. Parameter `keepAllThreads` controls what happens if image size is not a multiple of group size. Don't forget to set it to `true` if using barriers; in this case check image margins manually.

Define SRAM variables: `devSramVar`, `devSramMatrix`, `devSramArray` macros.
Make a barrier: `devSyncThreads()` macro.
Get the thread/group indices: `devThreadX`, `devThreadY`, `devGroupX`, `devGroupY` macros.

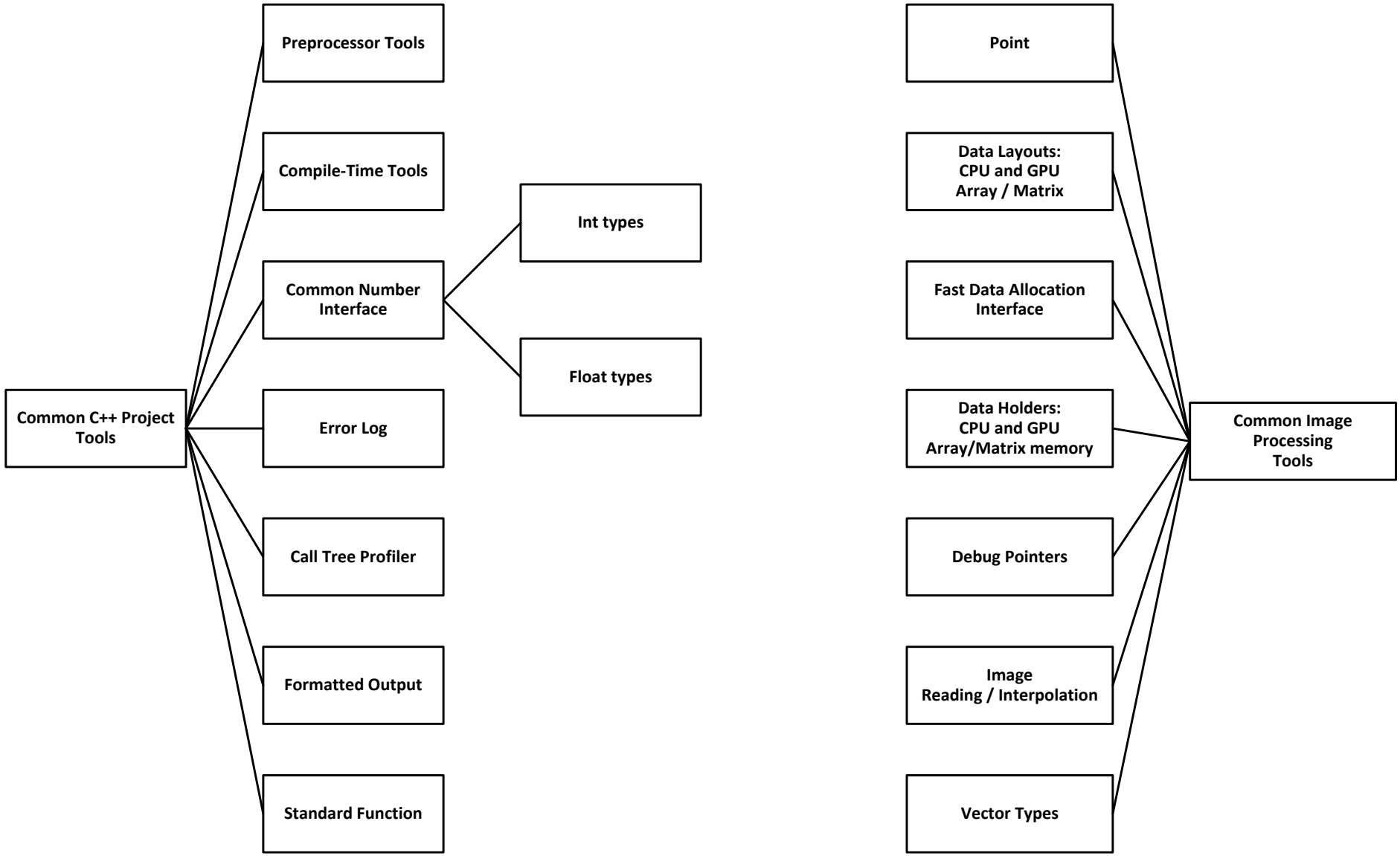
Appendix: Brief Project Structure



Appendix: GPU Compilation and Linking Support



Appendix: Basic Tools



Appendix: Algorithm Tools Abstraction Layer