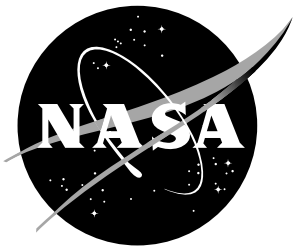
NASA/TM–2018–219897



# C++ Resource Intelligent Compilation for GPU Enabled Applications

*David J. Skudra*
*Universities Space Research Association, 615 National Ave, Mountain View, CA 94043*

*George E. Gorospe*
*SGT Inc., NASA Ames Research Center, Moffett Field, CA, 94035, USA*

April 2018

**NASA STI Program...in Profile**

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collection of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

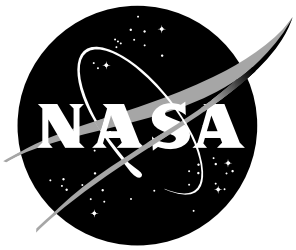
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at `http://www.sti.nasa.gov`
- E-mail your question to `help@sti.nasa.gov`
- Phone the NASA STI Information Desk at 757-864-9658
- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM–2018–219897



# C++ Resource Intelligent Compilation for GPU Enabled Applications

*David J. Skudra*
*Universities Space Research Association, 615 National Ave, Mountain View, CA 94043*

*George E. Gorospe*
*SGT Inc., NASA Ames Research Center, Moffett Field, CA, 94035, USA*

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, CA

April 2018

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

# Contents

# 1 Introduction

We are nearing the limits of Moore's Law [1] with current computing technology. As industries push for more performance from smaller systems, alternate methods of computation such as Graphics Processing Units (GPUs) should be considered. Many of these systems utilize the Compute Unified Device Architecture (CUDA) to give programmers access to individual compute elements of the GPU for general purpose computing tasks. Direct access to the GPU's parallel multi-core architecture enables highly efficient computation and can drastically reduce the time required for complex algorithms or data analysis. Of course not all systems have a CUDA-enabled device to leverage, and so applications must consider optional support for users with these devices. Resource Intelligent Compilation addresses this situation by enabling GPU-based acceleration of existing applications without affecting users without GPUs.

Resource Intelligent Compilation (RIC) creates C/C++ modules that can be compiled to create a standard CPU version or GPU accelerated version of a program, depending on hardware availability. This is accomplished through a toolbox of programming strategies based on features of the CUDA API. Using this toolbox, existing applications can be modified with ease to support GPU acceleration, and new applications can be generated with just a few simple modifications. All of this culminates in an accelerated application for users with the appropriate hardware, with no performance impact to standard systems. This memorandum presents all the important features involved in supporting RIC and an example of using RIC to accelerate an existing mathematical model, without removing support for standard users. Through this memorandum, NASA engineers can acquire a set of guidelines to follow for RIC-compliant development, seamlessly accelerating C/C++ applications.

A basic knowledge of CUDA and it's compilation pipeline is assumed, and some CMake familiarity.

# 2 Background

## 2.1 Prior GPU Work

Researchers at the NASA Ames Research Center have shown that many parallel applications implemented in CUDA outperform their CPU equivalents by multiple orders of magnitude [2]. In this research, parallelized Monte Carlo algorithms were used to demonstrate the application of GPUs towards computationally expensive algorithms and the expected performance increase. Furthermore, researchers hoped to leverage this technology to improve algorithm resolution (through more samples) and run-time. Simulation techniques like Monte Carlo algorithms are perfect candidates for GPU implementations, as they satisfy the optimal attributes within a CUDA program:

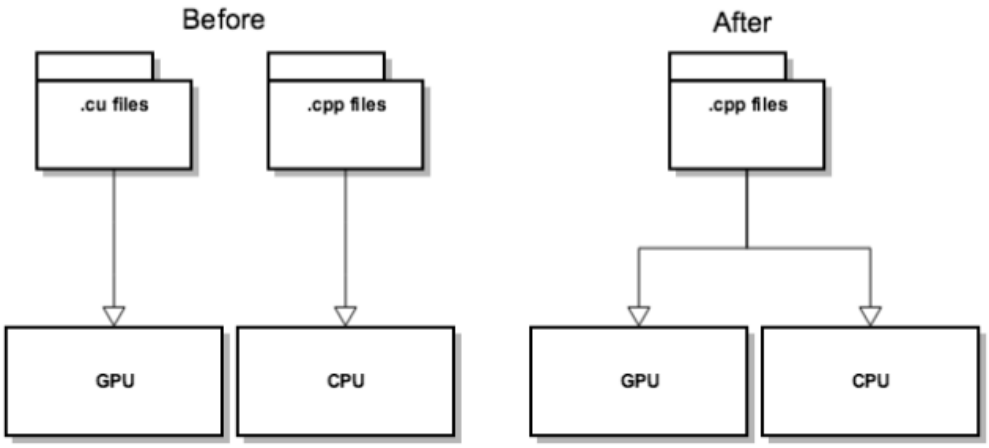
1. Algorithm operations may be easily parallelized
2. Parallelized operations follow the same instructions (no divergence)
3. Require an immense amount of resources to solve

## 2.2 Inspiration for RIC

Consider a case where a large pre-existing C++ library with thousands of users is found to be a good candidate for GPU acceleration, as it has computationally demanding components that can be easily parallelized. Pulling the rug out from the userbase and suddenly introducing CUDA-enabled devices as a hard requirement for using the library is not ideal, as not all users may possess the necessary hardware.

An alternative approach is to maintain a separate project that implements a GPU accelerated version of the demanding functions. The problem with this strategy is that this now requires additional support, as a second copy of the library must now be maintained and kept in stream with the original.

This problem is the inspiration for RIC. The ideal GPU accelerated program has all changes made inline with the original program, avoiding code duplication that requires extra maintenance, but at the same time does not impact users that don't have CUDA-enabled devices to leverage. RIC enables the acceleration of code within their original models by dynamically generating the code based on the available hardware. This is accomplished using a small set of tools and design strategies for seamless GPU acceleration.



# 3 RIC Toolbox

## 3.1 Preprocessor Definitions

As indicated in the background, RIC aims to dynamically generate the code needed to build a program based on the available hardware. This implies deciding what code to compile in advance, which can be accomplished using the preprocessor and different CUDA compilation pipelines. When compiling CUDA applications, the NVIDIA CUDA Compiler (nvcc) is used in place of gcc/g++. This allows applications to be built that include any CUDA host code (code executed on the CPU) and CUDA device code (code executed on the GPU).

Preprocessor definitions can be used to make code generation decisions, as the preprocessor resolves them before code is sent to the compiler. Fortunately, CUDA provides some useful definitions that can be used to help make these preprocessor decisions when compiled with nvcc. Below are two important preprocessor definitions, which serve similar purposes with a subtle difference between the two.

1. __CUDACC__

   This macro is used by source files executed **on the host** to determine if they are currently being compiled by nvcc. If it isn't being compiled by nvcc, it's left undefined which enables the definition to be used to dictate which blocks of code on the host should be compiled. This is very useful as most of the work with CUDA is not in writing the kernels themselves, but the setup and cleanup required outside of the kernel call.

   Below shows an example of when CUDACC is necessary, by using it to decide if device memory or host memory should be allocated for some histogram to be populated later.

```
// CUDA/C++ host code

// Code that is executed on the host but affects a possible
// kernel call
#ifdef __CUDACC__
  cudaMalloc((void **)&dayHistogram, sizeof(int) * timeHorizon));
  cudaMemset((void *)dayHistogram, 0, sizeof(int) * timeHorizon));
#else
  dayHistogram = (int *)calloc(timeHorizon, sizeof(int));
#endif
```

2. ___CUDA_ARCH___

   This macro is used to identify in **device code** what virtual architecture it is currently being compiled for. For RIC purposes, it is simply used as an indication of whether or not code is currently being compiled on a device, as it's otherwise undefined. Note that absolutely no host code can be dependent on the existence of ___CUDA_ARCH___.

   Below shows an example of when ARCH is necessary, using it to decide how to update a histogram that might be executed on the host or a device.

```
// CUDA/C++ device code

// Code that is executed on either the host or device
if (someCondition) {
#ifdef __CUDA_ARCH__
  atomicAdd(&dayHistogram[t], 1); // Avoid race conditions in threads
#else
  dayHistogram[t] += 1;
#endif
}
```

Formally, the difference between these two definitions is that ___CUDA_ARCH___ should exclusively be used to decide if code is being compiled to execute on a device, while ___CUDACC___ should be used to check what compiler is being used for the current source file. This is important as any CUDA program includes more than just the kernel that executes on the device, but the handler that is executed on the host. Simply put, decisions that will be made on the GPU must use CUDA_ARCH, while decisions made on the CPU must use CUDACC.

### 3.2 Qualifiers

Another set of critical tools used in RIC are function qualifiers. These are common tools used in most CUDA programs, but they are critical to RIC in order to support multiple compilation pipelines.

1. ___host___

   This qualifier is implicitly declared for any function that doesn't already own a qualifier. It simply states that the function is to be compiled for execution on the host. The existence of this qualifier is important later.

2. ___global___

   This qualifier indicates that a function is to be executed on the device and therefore must be put in the device compilation pipeline. Functions with this

qualifier can be invoked directly from host code, which is essentially where execution is transferred from host to device.

3. ___device___

   Device qualified functions are similar to global qualified functions in that they must be executed on the device, and compiled in the device pipeline. An additional requirement for device qualified functions is that they can only be invoked while already on the device (so within the scope of another device function, or a kernel function).

A neat trick with CUDA is that it allows functions to carry both a ___host___ and ___device___ qualifier at the same time, which puts a copy of the function into the host compilation pipeline and another into the device compilation pipeline. This is extremely useful in the context of RIC as this combined qualifier essentially eliminates the need to duplicate code that otherwise would require near identical computation steps. Combined, the definitions described earlier and these qualifiers create a powerful set of tools.

The example below shows how using a combination of preprocessor definitions and qualifiers, a few small modifications can be made to an existing CPU-implemented model to now support GPU accelerated. Of course, users who don't leverage a CUDA-enabled device won't see any difference while compiling this module as the preprocessor definitions protect their host compiler from any unfamiliar code.

```
// CUDA/C++ host OR device code

// Necessary to prevent qualifiers from getting in host pipeline
#ifdef __CUDACC__
  #define HOSTDEVICEQUALIFIER __host__ __device__
#else
  #define HOSTDEVICEQUALIFIER
#endif

HOSTDEVICEQUALIFIER void someModel(int particles) {
#ifdef __CUDA_ARCH__
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < particles; i += stride) {
#else
  for (int i = 0; i < particles; i++) {
#endif
  // 50 lines of common computation here
  }
}
```

Again, in order to support compilation on machines without CUDA, code such as the CUDA qualifiers cannot be present as it will cause errors. The preprocessor definitions described in the prior section are therefor used to create the proper definition, depending on what type of system the program is being compiled for. On a system with a CUDA-enabled device the HOSTDEVICEQUALIFIER is populated and therefor the function someModel is placed in both the GPU and CPU compilation pipelines. On a system without a CUDA-enabled device the HOSTDEVICEQUALIFIER is left blank, so it is ignored when resolved by the preprocessor.

### 3.3 Wrappers

In section 4.2, it was indicated that functions with the ___device___ qualifier prefixing them can only be called from other device or kernel functions. The simple solution here is to create a kernel function that simply passes on any necessary arguments to the host-device qualified code. This also means that a kernel function must be launched to call the device code, if being compiled for a GPU. This results in a second layer of wrapper functions that is required when using the GPU model. An example below shows example wrappers that would be required to launch a model in parallel using CUDA.

```
// CUDA/C++ host OR device code

#ifdef __CUDACC__
  #define KERNEL __global__
#else
  #define KERNEL
#endif

KERNEL void modelCallKern (int particles) {
  someModel(particles);
}

// Again, only exists in device compilation pipeline. Transfers program
// from host to device.
#ifdef __CUDACC__
void someModelGPU(int particles, int threadsPerBlock, int blocksPerThread)
    {
  modelCallKern<<<blocksPerGrid, threadsPerBlock>>>(particles);
}
#endif

void someModelCPU(int particles) {
  someModel(particles);
}
```

Using ___CUDACC___ it is ensured that compilation is unaffected on systems without CUDA-enabled devices. Above there are two nearly identical functions, someModelGPU and someModelCPU, which both cause someModel to eventually be invoked. The key difference is that a second wrapper, modelCallKern, exists for device compilation. It serves as the entry point to the device as it is prefixed with KERNEL, and is launched using CUDA notation, initiating all modelCallKern calls in parallel.

In summary, one of someModelGPU or someModelCPU will get invoked directly, but both end up making a call to someModel in their pipeline. The difference between the two is the extra layer of abstraction caused by someModelGPU launching a wrapper in parallel, with each parallel thread making a call to someModel.

# 4 Compilation

## 4.1 CUDA Compilation for RIC

An important component of RIC is the management of file extensions for source files. Files that include CUDA code are expected to have the ".cu" file extension, which can only be compiled by nvcc. Supporting multiple compilation modes presents a significant problem: if host-only compilation is to be supported, all the files must have ".cc" or ".cpp" as their file extension. In doing this, nvcc believes all source files to be standard C++ and won't compile for device usage. Fortunately, CUDA provides a compilation flag to get around this, -x, which indicates that the source file being compiled is to be treated as if it was a CUDA module and had the ".cu" file extension.

Example: A model that supports RIC exists as "someModel.cpp". The following two lines demonstrate how to compile someModel if only the CPU is to be used, and how to compile if there is a CUDA-enabled device is available.

```
g++ -c someModel.cpp // Compile for standard C++
nvcc -c -x cu someModel.cpp // Compile for CUDA
```

Using this feature, existing libraries can conveniently be left in their existing cpp/cc modules. At this point, a simple makefile could be created to build a RIC-compliant application. The only additional requirements would be to have two possible targets, one for a CPU build and the other for a GPU build, where the GPU target compiles using nvcc in place of gcc/g++, and the '-x cu' flag is passed.

## 4.2 Adding CMake Support with FindCUDA

Instead of using the same makefile and defining multiple targets for each form of compilation (CPU and GPU targets), it's worth utilizing CMake to generate buildsystems. CMake provides full support for CUDA, in the form of the FindCUDA toolset [3]. This means almost all of the legwork with CUDA is done for us,

and only a few extras are required to get CMake to support RIC.

The first step to incorporating CUDA to a CMake file is to add the FindCUDA project with a quick find_package call. This adds in a bunch of CMake functionalities and populates variables with useful CUDA related information.

```
# CMake code

# Grab the CUDA package
find_package(CUDA)
set(GPU_ACCELERATED ${CUDA_FOUND})
```

CMake provides a useful functionality where the variable CUDA_FOUND is automatically populated with the result of attempting to find CUDA on a system. This can be used hereafter for all decision related problems in the CMakeLists file that depend on whether or not a GPU can be leveraged. FindCUDA also populates numerous other variables, which get used for other commands [4]. A great example is CUDA_NVCC_FLAGS, which can be populated with any additional flags to pass to NVCC. This means it can hold the virtual architecture parameters to compile with, and more. Another useful flag is CUDA_PROPAGATE_HOST_FLAGS, a boolean value indicating if all host compiler flags (the ones passed to C++) should also be passed to NVCC. This can cause compilation issues, so it is best to disable flag passing and exclusively indicate what flags need to be turned on for NVCC.

```
# CMake code

set(CUDA_NVCC_FLAGS "${CUDA_NVCC_FLAGS} -gencode
    arch=compute_53,code=sm_53; -gencode arch=compute_53,code=compute_53;
    -std=c++11;")
set(CUDA_PROPAGATE_HOST_FLAGS off)
```

### 4.3 Setting up a project for RIC and CMake

Some modules in a project may only be necessary for specific compilation modes (GPU or CPU). Using some FindCUDA populated flags, it is possible to optionally include user created directories as needed. A good example of this is including a random number generation model that operates in parallel on GPUs, in place of using the standard <random> library. For a typical project that has 'inc' and 'src' directories setup, it's worth creating a new pair of directories 'incGPU' and 'srcGPU' to store all GPU-only modules.

```
|____build
|____CMakeLists.txt
|____inc
| |____callModels.h
|____incGPU
| |____prng.h
| |____debugCFP.h
|____src
| |____callModels.cpp
| |____main.cpp
|____srcGPU
| |____prng.cu
| |____debugCFP.cu
```

Continuing with this example, GPU only files prng.cu and prng.h, would be stored in the GPU directories and then only included if the variable from the previous section, GPU_ACCELERATED, was defined.

```
# CMake code

# Add directories
include_directories(${CMAKE_CURRENT_SOURCE_DIR}/inc/)
if (${GPU_ACCELERATED})
  include_directories(${CMAKE_CURRENT_SOURCE_DIR}/incGPU/)
endif()

# Setup environments, depending on GPU accel. status
set(SRCS src/main.cpp src/callModels.cpp)
set(INCS inc/callModels.h)

if (${GPU_ACCELERATED})
  set(SRCS ${SRCS} srcGPU/prng.cu)
  set(INCS ${INCS} incGPU/prng.h)
endif()
```

Another important feature described in section 5.1 was how to compile files with the 'cpp' extension as CUDA files in place of C++. This is accomplished using the '-x cu' compilation option when generating the binary for a source file. With CMake, this option can't be passed directly through CUDA_NVCC_FLAGS, and instead set_source_file_properties must be used. This way when compilation time comes around, the file extension is ignored and, source files such as callModels.cpp can be treated as CUDA sources.

```
# CMake code

if (${GPU_ACCELERATED} AND ${GPU_SUPPORTED})
  set_source_files_properties(${CMAKE_CURRENT_SOURCE_DIR}/src/callModels.cpp
                              PROPERTIES CUDA_SOURCE_PROPERTY_FORMAT OBJ)
endif()
```

This feature was added in CMake version 3.3.0 (2015-07-23 release) [5], so RIC requires a modern version of cmake. The additional variable, 'GPU_SUPPORTED' above, gets manually populated as a boolean value indicating whether or not the version of CMake used is 3.3.0 or above. This is because the '-x cu' compile option to compile cpp files as cu files isn't supported in lower versions of CMake.

The final requirement of CMake is to indicate how to actually make the executable. FindCUDA has its own version of add_executable, named cuda_add_executable. It operates in the same manner as it's predecessor, but instead uses NVCC. Note that cuda_add_executable will still attempt to use g++ when compiling modules unless they have the CUDA_SOURCE_PROPERTY indicated, which is enabled by default for '.cu' files, or enabled manually as performed in this section.

```
# CMake code

if (${GPU_ACCELERATED} AND ${GPU_SUPPORTED})
  # Create GPU executable and link CUDA library
  cuda_add_executable(stockModel ${SRCS} ${INCS})
  target_link_libraries(stockModel -L/usr/local/cuda/lib64 -lcurand)
else()
  # Create executable
  add_executable(stockModel ${SRCS} ${INCS})
endif()
```

As before, the flags set to determine what target is being compiled for are used to decide if add_executable is used, or it's CUDA equivalent. An extra line is also used to link a standard CUDA library, 'curand'. This can also be extended to any other members of the standard library such as cudart. FindCUDA operates in an interesting manner as it verifies the existence of the CUDA standard library when find_package is initially called, but doesn't add it to the known paths for the linker to search through. Hence, it must be manually indicated for any CUDA libraries to be linked.

# 5 Results

## 5.1 Example Software: Stock Price Model

Most of the code samples provided throughout this memorandum have been lifted from a Monte Carlo simulation involving stock prices created as a demonstration of RIC properties.

The sample code uses the Monte Carlo method to simulate stock prices over time, attempting to come up with an approximate amount of time in which a specific price boundary (upper or lower) on the stock is hit. Because this simulation uses the Monte Carlo method, it is a prime candidate for GPU acceleration by making the numerous samples in the simulation operate in parallel. Of course, this simulation might still need to support systems which do not have CUDA-enabled devices, making RIC design principles crucial.

## 5.2 Performance

The culmination of all previous sections is that the application has been seamlessly updated to support accelerated compilation, without abandoning architectures by introducing a new hard dependency. Testing for the stock model is conducted on an NVIDIA Jetson TX1 (Maxwell GPU with 256 CUDA cores), and a 2013 Macbook Air (Haswell i7 @ 1.7 GHz). Below are the results of running the same program on the two different systems; the former with a CUDA-enabled device, the latter without.

| Macbook Air 2013, Intel i7 1.7 GHz | NVIDIA Jetson TX1 (CUDA enabled) |
|---|---|
| $ cd build<br>$ cmake ..<br>$ make<br>$ ./stockModel 32 128 1000000 1 200 | $ cd build<br>$ cmake ..<br>$ make<br>$ ./stockModel 32 128 1000000 1 200 |
| Calling CPU version of model!<br>Mean day boundary hit: 84<br>Duration: 8.36235 | Calling GPU version of model!<br>Mean day boundary hit: 84<br>Duration: 0.003602 |

The most important take away from these results, aside from the clearly superior performance of a GPU accelerated program, is that the pipeline from compilation to output is the exact same. RIC has permitted seamless introduction of GPU acceleration as the 1st column, a system without a CUDA-enabled device, is still able to build and run the same application without issue.

# 6 Summary

Using RIC, many existing applications can easily be modified to support GPU accelerated programs, with no impact whatsoever to a standard user who doesn't have a CUDA-enabled device to leverage. This methodology only requires a small fixed initial cost when modifying existing application and has next to no variable cost as the application is updated over time. New applications can also be created with

ease that support RIC using the toolbox described in this memorandum.

This memorandum covered a brief background on GPU acceleration, the inspiration for RIC, the toolbox that makes this design paradigm possible, and results from applying it to an existing software model.

## References


1. R. R. Schaller, "Moore's Law - Past, Present, and Future", IEEE Spectrum, June 1997, pp. 52-59

2. G. E. Gorospe, "GPU Accelerated Prognostics", October 2017, https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20170012211.pdf

3. FindCUDA Documentation https://cmake.org/cmake/help/v3.0/module/FindCUDA.html

4. FindCUDA Flags https://cmake.org/cmake/help/v3.0/module/FindCUDA.html

5. CMake 3.3 https://github.com/Kitware/CMake/blob/v3.3.0/Modules/FindCUDA.cmake#L1311

| REPORT DOCUMENTATION PAGE | Form Approved<br>OMB No. 0704–0188 |
|---|---|



| 1. REPORT DATE (DD-MM-YYYY) | 2. REPORT TYPE | 3. DATES COVERED (From - To) |
|---|---|---|
| 01-04-2018 | Technical Memorandum | |

| 4. TITLE AND SUBTITLE | 5a. CONTRACT NUMBER |
|---|---|
| C++ Resource Intelligent Compilation for GPU Enabled Applications | 5b. GRANT NUMBER |
| | 5c. PROGRAM ELEMENT NUMBER |



| 6. AUTHOR(S) | 5d. PROJECT NUMBER |
|---|---|
| David J. Skudra, George E. Gorospe | 5e. TASK NUMBER |
| | 5f. WORK UNIT NUMBER |

| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) | 8. PERFORMING ORGANIZATION REPORT NUMBER |
|---|---|
| NASA Ames Research Center<br>Moffett Field, CA | |



| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | 10. SPONSOR/MONITOR'S ACRONYM(S) |
|---|---|
| National Aeronautics and Space Administration<br>Washington, DC 20546-0001 | NASA |
| | 11. SPONSOR/MONITOR'S REPORT NUMBER(S)<br>NASA/TM–2018–219897 |

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified-Unlimited
Subject Category 64
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES

An electronic version can be found at http://ntrs.nasa.gov.
Point of contact: dskudra@gmail.com or george.e.gorospe@nasa.gov

15. SUBJECT TERMS

RIC, CUDA, C++, GPU

| 16. SECURITY CLASSIFICATION OF: | | | | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | | c. THIS PAGE | | | STI Information Desk (help@sti.nasa.gov) |
| U | U | U | | 20 | 19b. TELEPHONE NUMBER (Include area code)<br>(757) 864-9658 |