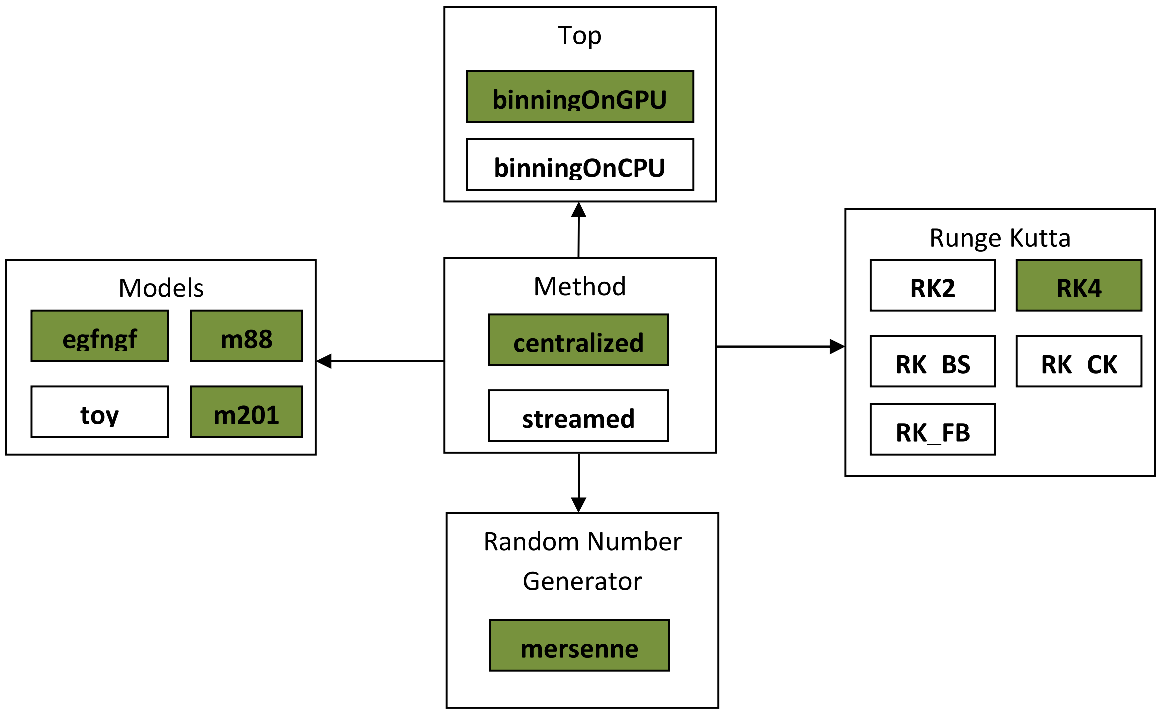
# PADA-GPU documentation

## 1. Code structure and compilation configuration

The implementation is modular. The configuration is pinned at compile time. The following drawing represents the relationships between the available modules. The marked modules (green background) are those used for the Bioinformatics submission, and they were extensively tested.



To compile, one needs to run the command below. The compilation process configures which modules are utilized. The default values are those assigned to each parameter in the example below, and they are also assigned if the correspondent *make* parameter is not specified.

```
make MODEL=egfngf TOP=binningOnGPU METHOD=centralized RKmethod=RK4 ARCH=sm_20 REAL=double
```

1.1. The **Model**s (*models/*.model*) were documented in the ICPP submission. Among the four, *m88* is the largest, and takes a significant compilation and execution time. *egfngf* and *m201* are comparable in terms of run-time and compilation performance. The models are represented in a special format which allows their contents to be handled by the C preprocessor.

1.2. The implementation's **Top** module (*binningOn[G/C]PU/top.cu, binningOn[G/C]PU/incBins.cu*) orchestrates the execution among a number of GPUs. Two versions are available. They differ in terms of communication requirement based on how the data is collected / binned (details in Section 2.1). The list of utilized GPUs can be edited in *binningOn[G/C]PU/config_multigpu.h*. The list contains three fields for each GPU: (1) the GPU index on the respective host, (2) the host name (can be localhost), and (3) the working folder on the remote host. Key-based authentication (without an interactive password) is required to connect automatically to remote hosts. Files will be written in the remote working folder specified. The final results are written on the master host by *binningOn[G/C]PU/WriteFiles.c.*

1.3. The **method** manages how parallel iterations are executed locally. Two versions are available which differ on how they handle the storage of intermediate data (details in Section 2.2). For the current implementation, the *centralized* method (*centralized/simulation.cu*) appears to be faster, but this method scales worse because it requires all the intermediate data to be stored together in the shared memory. The *streamed* method (*streamed/simulation.cu)* releases this constraint, and allows subsets of the data to be present in the shared memory.

1.4. The **Runge Kutta** algorithm handles one integration step. Several versions of the algorithm are implemented (details in Section 2.3). They differ in terms of integration precision. A forth order algorithm has been used for the ICPP paper (*RK4/rk.cu*). Some of the versions are adaptive (*RK_FB/rk.cu* and *RK_CK/rk.cu*).

1.5. Older GPU **arch**itectures (<2.0) do not allow certain features. The compilation may be directed to use a different architecture such as *sm_13* or *sm_10.*

1.6. The precision of the **real** types utilized can be *float* or *double*. The latter provides more accurate results, but doubles the storage size. This limits the total amount of data that can reside at any time in shared memory, hence it can affect scalability.

## 2. Module details

2.1. The ***top.cu*** file contains the *main* function. This function behaves differently for a master process and a slave process in a multi-GPU run. The user launches the master process from the command line. The remaining slave processes are launched via *ssh* (if they are remote). The master collects the resulting data files from both the remote (via *scp*) and the local slaves. The entire process is automatic, if the interactive logon is prevented as explained above.

The two versions of *top.*cu differ in the way the results are collected for binning. The binning process requires each trajectory to increment a few locations in a large matrix. As the trajectories are computed on different GPUs, the strategy can be:

a) The binning is done on the GPU (*binningOnGPU)*, and the resulting matrices are combined; this requires the transfer of the entire matrix over *scp,* followed by their combination, element by element (lines 141-153).

b) The binning is done on the CPU (*binningOnCPU)*; each host transfers a list of pointers to the locations it needs to implement. The transfer size will depend on the number of computed trajectories, but may reduce the overhead for large applications that compute a small number of trajectories (158-170).

Besides the top file, the binning behavior is also altered in the kernel. The kernel accomplishes the binning through a function *incBin(*in *incBins.cu)* which receives an offset to the table where it increments. In (a) the binning increments an element in the table, while in (b) It stores the pointer to the table element.

Eventually, the output of the results is done through code available in *WriteFiles.c*.

2.2. a) The **centralized** method instantiates as many parallel simulations as they fit in the shared memory. However, this number is not determined automatically, but it is provided by the user. The *simulation* function initializes the kernel with PAR_SIMS simulations. It allocates a number of locations in shared memory: NR_CONSTS + NR_RECS + 3 * NR_VARS + NR_AVARS.

These correspond to constants, reciprocals (computed only once), variables, deltas, and fat variables.

The *kernel* allocates additional global memory for the binning procedure. This memory is accessed seldom, so it will not affect the performance. It will run in parallel with PAR_SIMS threads distributed such that WORKERS_PER_IT threads correspond to a single trajectory. The identical threads corresponding to different trajectories are adjacent in the indexing.

The random number generator is initialized in each thread through *initializeGenerator*. The subsequent loop will be active for the entire simulation group. Once the variations are computed for all the variables of each trajectory, I*ncrements* is invoked to process these differences and add them to the current state of the variables. Once a binning point is reached, *BinPackVars* and *BinPackConsts* are invoked to bin the data.

The distribution among the WORKERS_PER_IT threads is accomplished statically, based on a heuristic distribution based on the execution time of each operator. This is done through the C preprocessor. As the model can be preprocessed to any desired form through the C preprocessor, it is transformed into a quantitative representation which reflects the weight of each equation. Using macros, the equations are visited and the included operators are substituted by a set of numbers which can be added.

In this context, a constant, *current_weight*, stores the total weight of the visited equations (variable which has a constant value at a given point in the program, it can be derived through constant propagation). Because this substitution is done using the preprocessor, the value of *current_weight* can be tracked statically during compilation. Using a second macro substitution, a set of statically defined *if* clauses is generated, and these clauses determine how the equations are executed in the WORKERS_PER_IT threads, based on the value of *current_weight* as available before each *if* clause.

b) The **streamed** method uses a java function to generate an equation schedule. This schedule contains a set of phases. separated by synchronization. In each phase, a set of unused locations in shared memory is filled with new values, which belong to the variables stored in global memory. This is done in parallel with a set of computations on the untouched variables in shared memory. Two files are generated, *declaration.cu* and *orchestration.cu,* which are integrated instead of *simulation.cu.*

2.3. The **Runge Kutta *(***rk.cu) is a macro utilized by the equation instantiation. Based on which macro vesion is utilised, the computation transparently replaces one version of RK with another. RK2 and Rk4 do not have side effects, and simply divide the integration interval into a set of subintervals over which they do precision integration. However, the more complex versions of RK do have side effects, as they affect whether the integration time interval is too large. Hence, the macro receives the time interval variable, which may be altered inside it.

## 3. Macro replacement

Macro replacement using C preprocessor has been used to prevent an additional level of indirection. Instead of interpreting the model from a table, the code itself reflects the model. In order to do this, the model is included at different points throughout the code. Each inclusion, however, is preceded by a set of macro definitions.

3.1. *WriteFiles.c* - each model equation is transformed into a file printing routine. This takes into account that there is one equation for each variable, and the table data for each variable goes to a different file.

3.2. *GenRandom/simulation.cu* - each constant and equation is replaced by an assignment to the shared memory location. This assignment contains a call to *extractNumber*, a function which provides a random number

3.3. The computation of *total_weight* in *simulation.cu*. The variables from the equations are ignored, and instead the operators are redefined to numbers. All these numbers are added together by the preprocessor and assigned to the constant *total_weight.*

3.4. *RungeKutta/simulation.cu* - based on the previously computed *total_weight* it is possible to determine how to distribute the computation among a set of threads. The computation is encapsulated in an *if* clause, which will check the partial weight of the equations visited until that point, and compare it with the range of weights accepted by the current thread. These *if* clauses are resolved statically by the compiler, as they involve only constants.

*3.5. FatNodes/simulation.cu* - the same as above, however, this distributes only the formula for the fat nodes. For simplicity, the equation count instead of weight is used, as the fat nodes computation appears only each 100 simulation steps.

*3.6. IncBins, BinPackVars, Increments/simulation.cu* - once the delta values of each variable are computed, they each need to be added to the corresponding equation. The information about the equation is extracted by converting each model equation into a set of addtions.

## 4. Configuration, execution and verification

4.1. The *centralized* version requires the manual setup of a few parameters, available in *config/*.centralized.* These parameters are:

- PAR_SIMS the number of parallel simulations; if too large, the shared memory is exhausted
- WORK the number of compute threads per simulation trace
- ACCESS the number of memory access threads per simulation trace
- DIV / MUL / PLUS / MINUS the weight of the operators on the current architecture

4.2. The result of the *make* command is a new folder, placed inside the *bin* folder; its name is composed of a concatenation of all the *make* settings. This folder contains the executable, and will also contain the results of the execution. For execution, the number of traces has to be specified as a parameter. Once the execution has completed, the results can be verified by running the corresponding version of the validation application. This application also needs a parameter, which is the relative path of the folder where the execution results are stored.

```
./code TRACES           # runs the specified number of TRACES on GPU 0 on current machine
./code TRACES -gpu N     # runs the specified number of TRACES on GPU N on current machine
./code TRACES -master conf_file # runs as master distributing TRACES onto several GPUs
```