Developing on DualSPHysics:

examples on code modification and extension

O. García-Feal [orlando@uvigo.es], L. Hosain, J.M. Domínguez, A.J.C. Crespo

Universidade Vigo

Environmental | Physics | Laboratory
EPhysLab

cpu gpu
DualSPHysics

4th DualSPHysics Users Workshop · Instituto Superior Técnico, University of Lisbon 22nd-24th October 2018

Outline of this presentation

1. Where you can download the code from?
2. Obtaining documentation
3. How to modify the code: Example of heat transfer
    1. Download code from GITHUB
    2. Implementation for CPU
        1. Add new variables
        2. Add new equation (particle interaction)
        3. Update new variables in time
        4. Use new var. in post-processing tools
    3. Implementation for GPU
        1. Add new variables
        2. Add new equation (particle interaction)
        3. Update new variables in time

0. How to use this presentation

This presentation was designed as a guide on how to modify DualSPHysics.

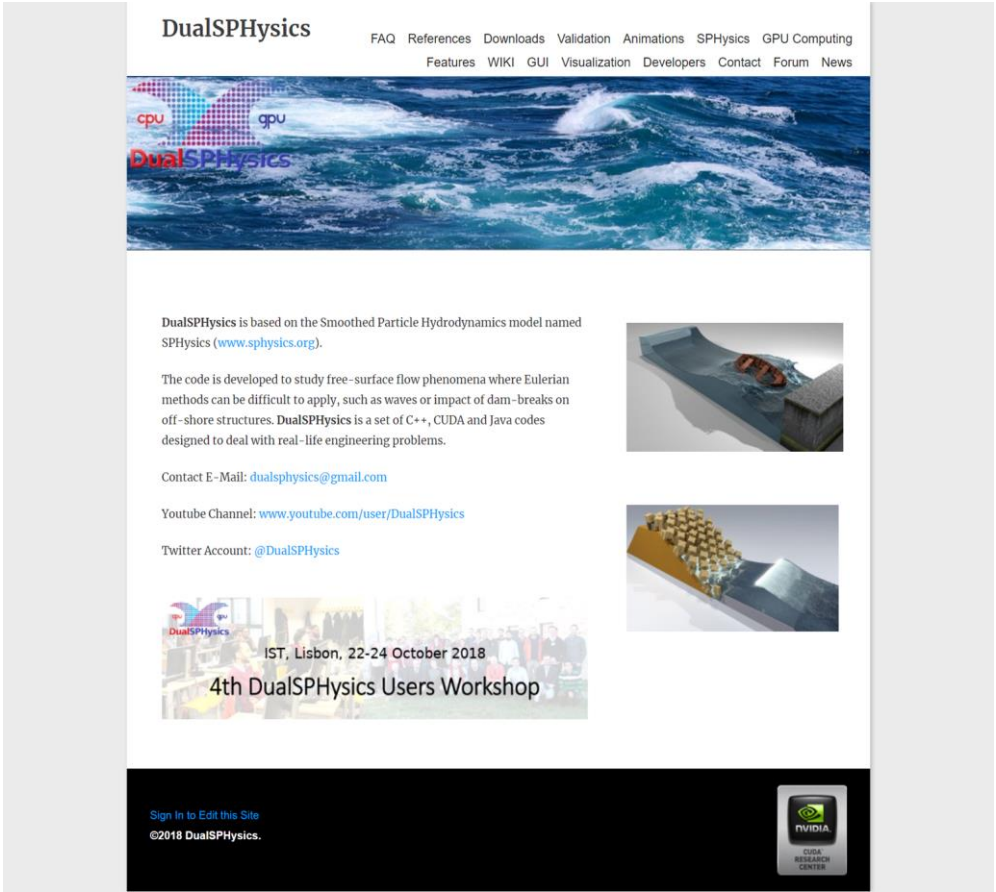
So you can download it and follow step-by-step at your home.

You can download it right now from the workshop website!

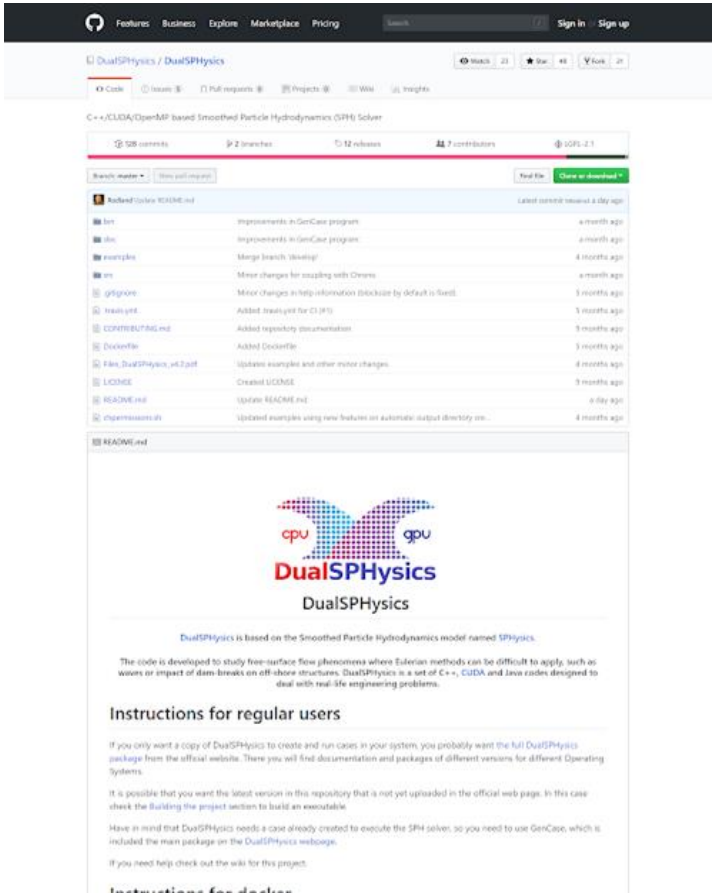
So don’t be afraid if you see lots of code and enjoy ☺

1. Where can be the source code downloaded from?

Full DualSPHysics package:



DualSPHysics GitHub Repository:



Full DualSPHysics package:

- Latest stable version.
- Includes all the documentation.
- Includes lots of examples.

Better to learn and run simulations.

DualSPHysics GitHub Repository:

- Includes latest developments.
- **You can merge DualSPHysics updates into your code!**
- You can make pull requests to DualSPHysics.

Better to develop.

2. Obtaining documentation

DualSPHysics wiki: https://github.com/DualSPHysics/DualSPHysics/wiki

PDF Guides of the oficial package: `./doc/guides`

Command line options: `./doc/help` or `<binary> -help`

XML Templates: `./doc/xml_format`

Example cases: `./examples`

Code documentation: http://dual.sphysics.org/doxygen

General documentation

C++: https://en.cppreference.com/w/

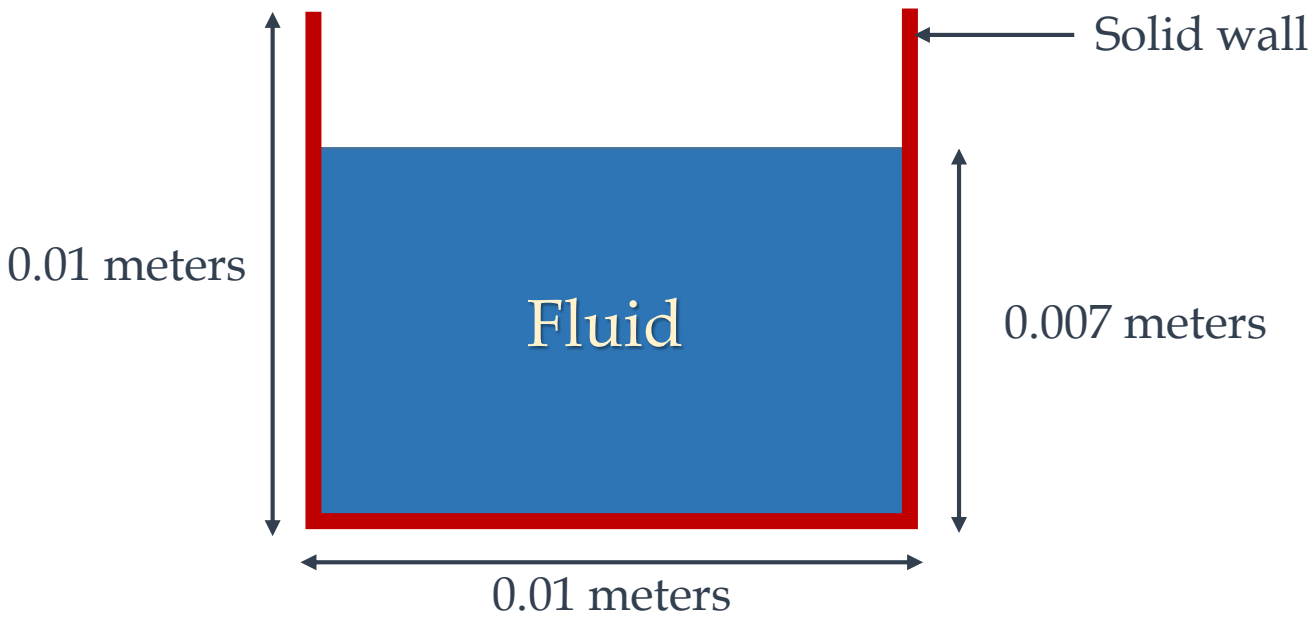
Debuging in Visual Studio: https://docs.microsoft.com/en-us/visualstudio/debugger/

Nvidia CUDA: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

GIT: https://git-scm.com/book/en/v2

General programming questions: https://stackoverflow.com/

3. Example of heat transfer



The solid wall transfers heat to the fluid.

The temperature of the fluid is 293 K

The temperature for the walls is constant.

The temperature of the walls is 363 K

It is very important to start defining a simple case!

Defining the ecuations

Energy conservation equation in its SPH form (Monaghan, 2005):

$$c_{p,a}\frac{\partial T}{\partial t} = \sum_b \frac{m_b}{\rho_a \rho_b}\frac{4k_a k_b}{k_a + k_b}(T_a - T_b)\frac{\nabla_a W_{ab}}{r_{ab}}$$

- c_p Specific heat capacity (constant)
- T Temperature
- m Mass (constant)
- ρ Density
- k Thermal conductivity (constant)
- W_{ab} Kernel function
- r_{ab} Distance

SPH Formulation

Momentum Equation:

$$\frac{dv_a}{dt} = -\sum_b m_b \left(\frac{P_b + P_a}{\rho_b \cdot \rho_a} + \Pi_{ab} \right) \nabla_a W_{ab} + g$$

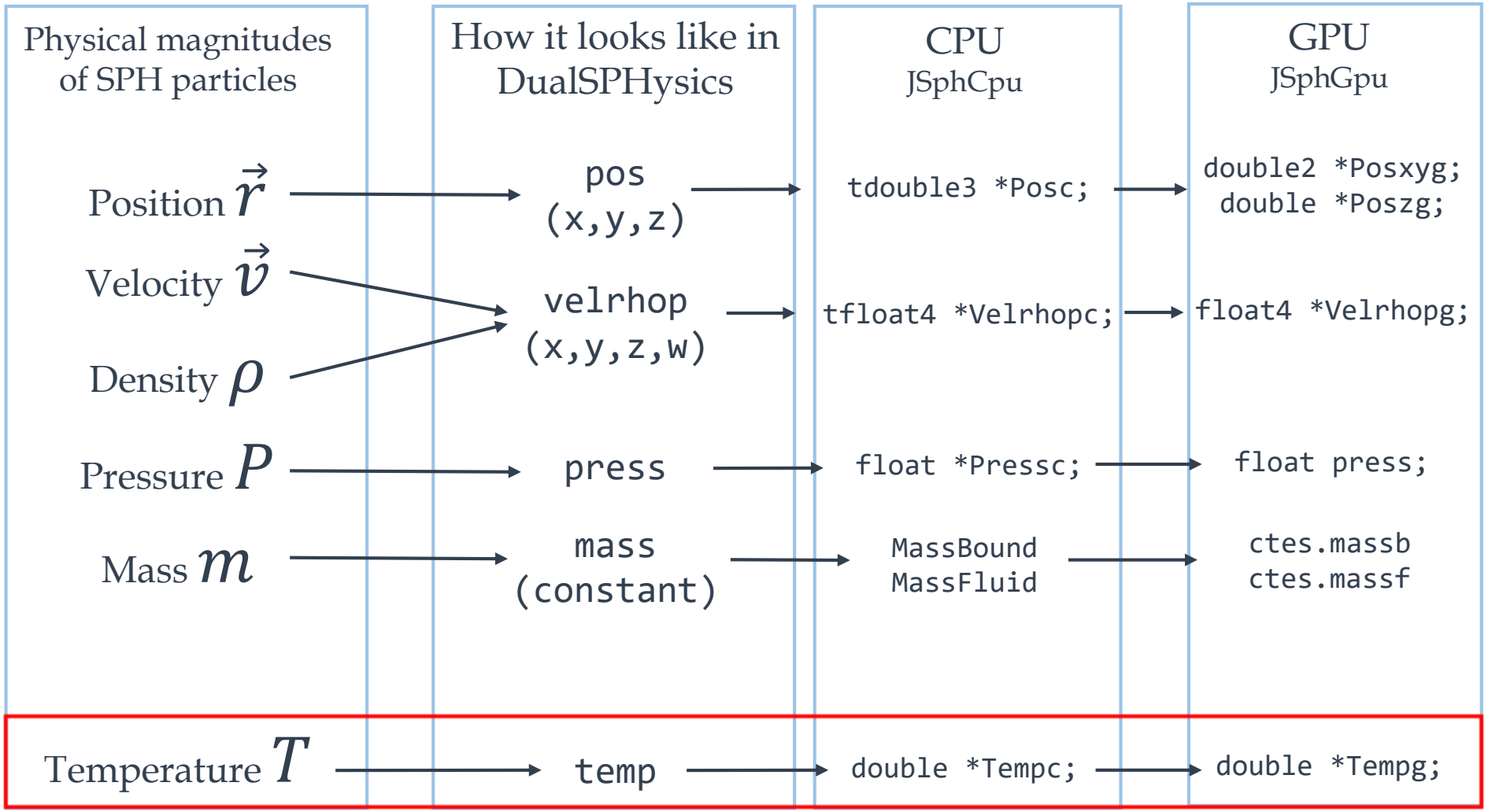
Continuity Equation:

$$\frac{d\rho_a}{dt} = \sum_b m_b v_{ab} \cdot \nabla_a W_{ab}$$

Temperature Equation:

$$c_{p,a} \frac{\partial T}{\partial t} = \sum_b \frac{m_b}{\rho_a \rho_b} \frac{4 k_a k_b}{k_a + k_b} (T_a - T_b) \frac{\nabla_a W_{ab}}{r_{ab}}$$

DualSPHysics variables nomenclature



Updating particles

DualSPHysics includes a choice of numerical integration schemes like Verlet and Symplectic.

But simplifying, the new value for a variable is computed as follows:

$$\vec{r}^{n+1} = \vec{r}^{n} + \frac{d\vec{r}}{dt} \cdot \Delta t \longrightarrow \frac{d\vec{r}}{dt} = \vec{v}$$

$$\vec{v}^{n+1} = \vec{v}^{n} + \frac{d\vec{v}}{dt} \cdot \Delta t \longrightarrow$$ `tfloat3 *Acec;` `float3 *Aceg;`

$$\vec{\rho}^{n+1} = \vec{\rho}^{n} + \frac{d\vec{\rho}}{dt} \cdot \Delta t \longrightarrow$$ `float *Arc;` `float *Arg;`

$$\vec{T}^{n+1} = \vec{T}^{n} + \frac{d\vec{T}}{dt} \cdot \Delta t \longrightarrow$$ `double *Atempc;` `double *Atempg;`

DualSPHysics source files

Common

Functions (.h .cpp)
FunctionsMath (.h .cpp)
JDsphConfig (.h .cpp)
JException (.h .cpp)
JLog2 (.h .cpp)
JMatrix4.h
JMeanValues (.h .cpp)
JObject (.h .cpp)
JParticlesDef.h
JPeriodicDef.h
JRadixSort (.h .cpp)
JRangeFilter (.h .cpp)
JReadDatafile (.h .cpp)
JSaveCsv2 (.h .cpp)
JSpaceCtes (.h .cpp)
JSpaceEParms (.h .cpp)
JSpaceParts (.h .cpp)
JSpaceProperties (.h .cpp)
JTimeControl (.h .cpp)
JTimer.h
JTimerClock.h
JXml (.h .cpp)
TypesDef.h

Common_BinaryFiles
JBinaryData (.h .cpp)
JPartDataBi4 (.h .cpp)
JPartDataHead (.h .cpp)
JPartFloatBi4 (.h .cpp)
JPartOutBi4Save (.h .cpp)

Common_Gpu
FunctionsCuda (.h .cpp)
FunctionsMath_ker.cu
JObjectGpu (.h .cpp)
JReduSum_ker (.h .cu)
JTimerCuda.h

Common_LibJFormatFiles2
JFormatFiles2.h
JFormatFiles2_x64.lib / libjformatfiles2_64.a

Common_LibJWaveGen
JWaveGen.h
JWaveOrder2_ker (.h .cu)
JWaveSpectrumGpu (.h .cpp)
JWaveGen_x64 .lib / libjwavegen_64.a

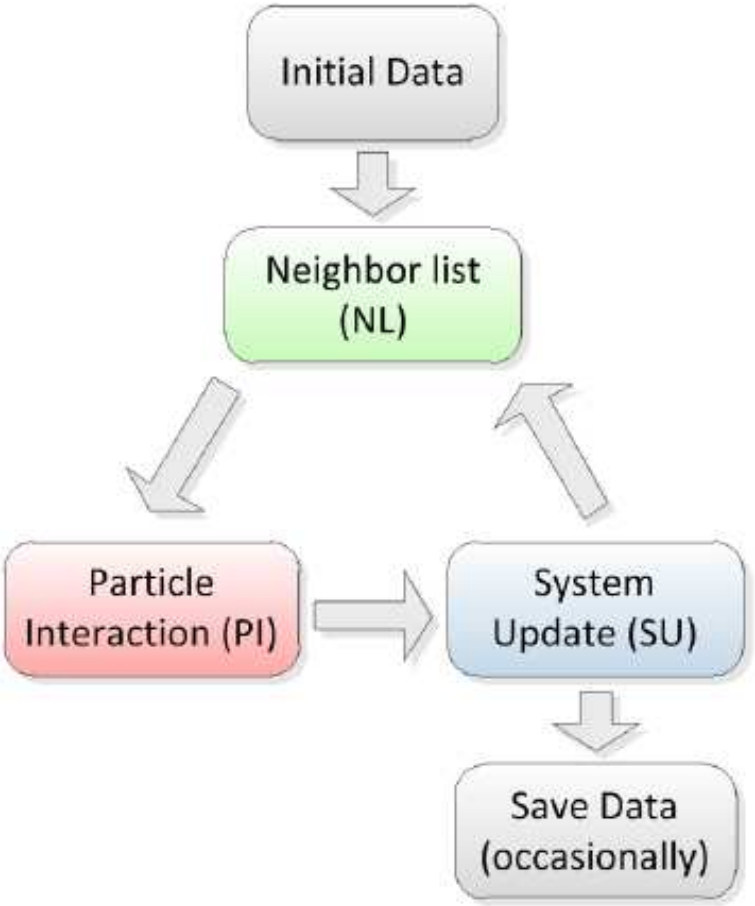
Common_Motion
JMotion (.h .cpp)
JMotionEvent (.h)
JMotionList (.h .cpp)
JMotionMov (.h .cpp)
JMotionObj (.h .cpp)
JMotionPos (.h .cpp)

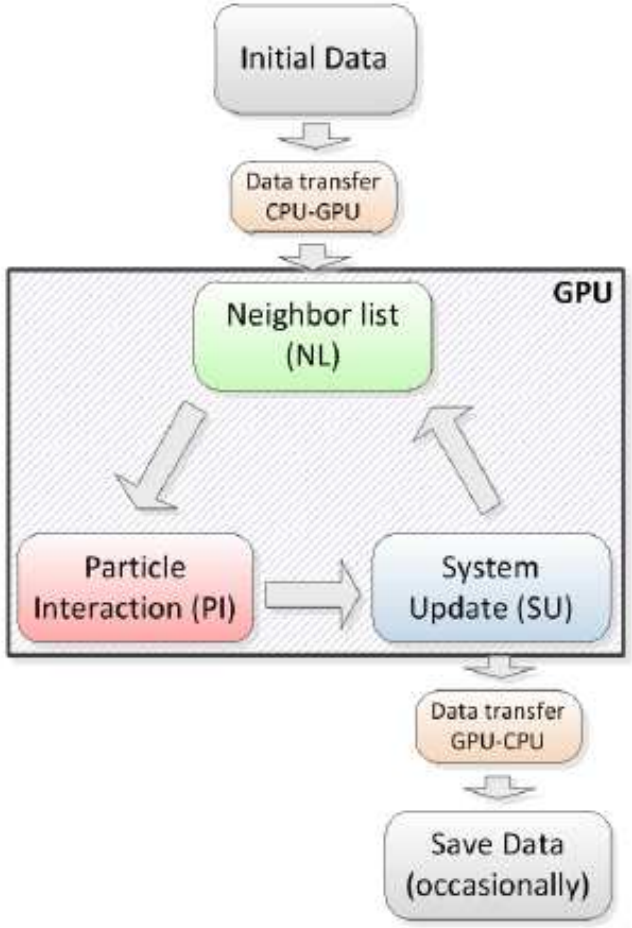
SPH on CPU	SPH on GPU
main.cpp JCfgRun (.h .cpp) JDamping (.h .cpp) JGaugeItem (.h .cpp) JGaugeSystem (.h .cpp) JPartsLoad4 (.h .cpp) JPartsOut (.h .cpp) JSaveDt.cpp (.h .cpp) JSph (.h .cpp) JSphAccInput (.h .cpp) JSphDtFixed (.h .cpp) JSphInitialize (.h .cpp) JSphMk (.h .cpp) JSphMotion (.h .cpp) JSphVisco (.h .cpp) JTimeOut (.h .cpp) OmpDefs.h Types.h	
JArraysCpu (.h .cpp) JCellDivCpu (.h .cpp) JCellDivCpuSingle (.h .cpp) JSphCpu (.h .cpp) JSphCpuSingle (.h .cpp) JSphTimersCpu.h	JArraysGpu (.h .cpp) JBlockSizeAuto (.h .cpp) JCellDivGpu (.h .cpp) JCellDivGpu_ker (.h .cu) JCellDivGpuSingle (.h .cpp) JCellDivGpuSingle_ker (.h .cu) JGauge_ker (.h .cu) JSphGpu (.h .cpp) JSphGpu_ker (.h .cu) JSphGpuSingle (.h .cpp) JSphTimersGpu.h

Source files that we will modify

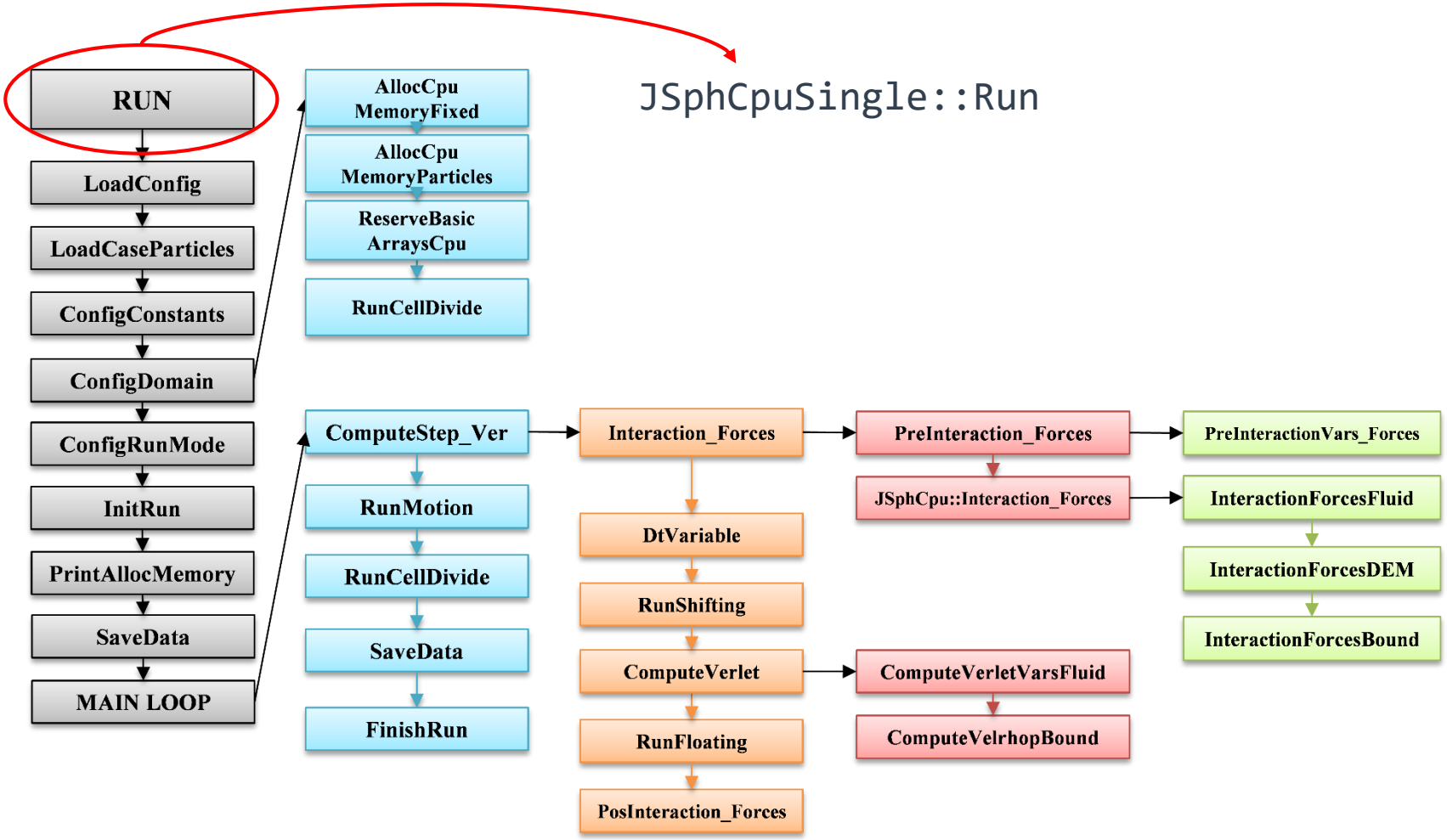
Common	`JSph.h/cpp`	Attributes and functions shared by CPU & GPU simulations.
CPU	`JSphCpu.h/cpp`	Attributes and functions used only by CPU simulations.
	`JSphCpuSingle.h/cpp`	Attributes and functions used to arrange CPU simulations.
	`JCellDivCpu.h/cpp`	Generates neighbor list for CPU simulations.
GPU	`JSphGpu.h/cpp`	Attributes and functions used only by GPU simulations.
	`JSphGpu_ker.h/cu`	CUDA kernels for GPU simulations.
	`JSphGpuSingle.h/cpp`	Attributes and functions used to arrange GPU simulations.
	`JCellDivGpu.h/cpp`	Generates neighbor list for GPU simulations.
	`JCellDivGpu_ker.h/cu`	CUDA Kernel for GPU neighbor generation.

DualSPHysics general simulation flow



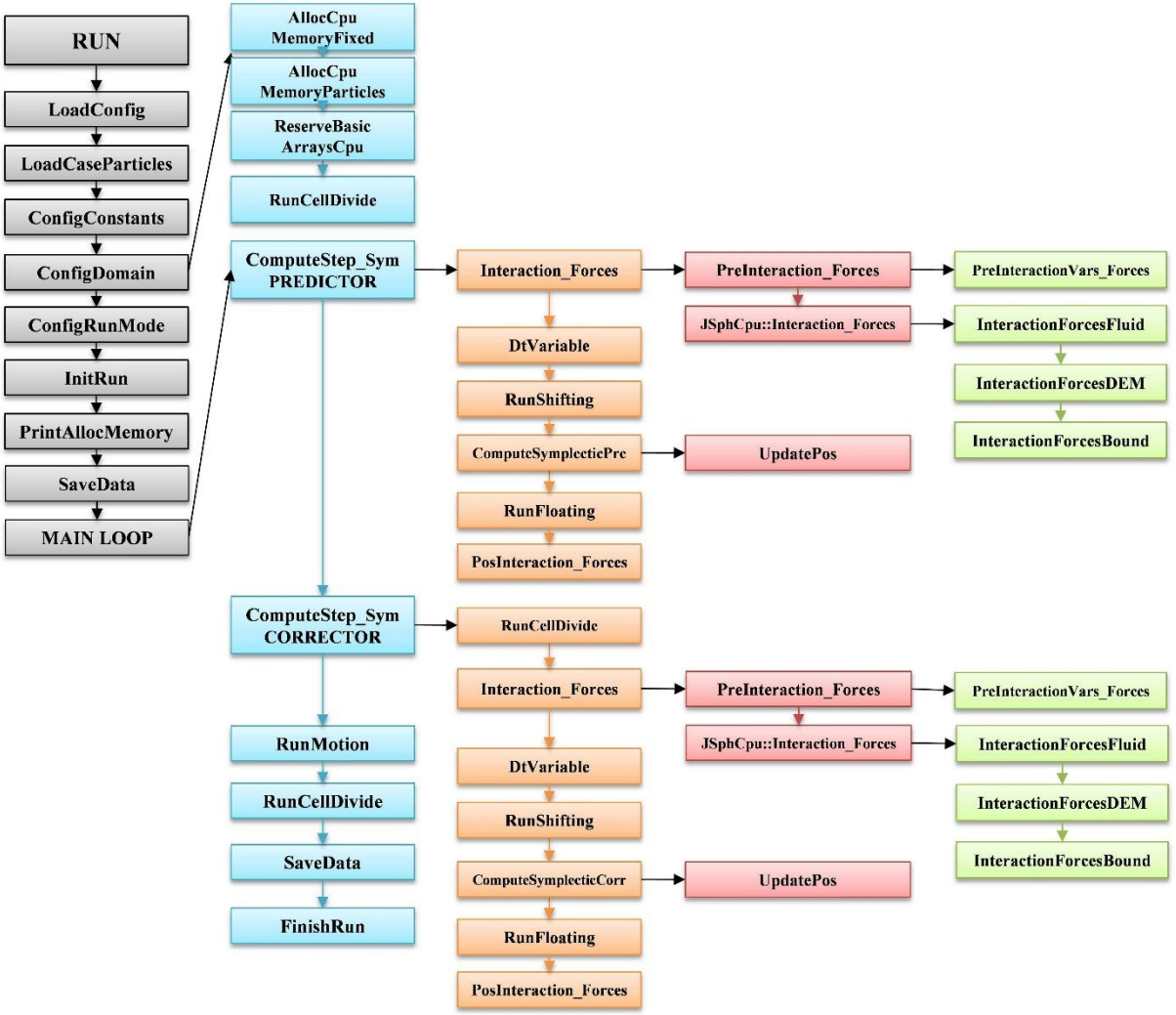


DualSPHysics simulation run-flow



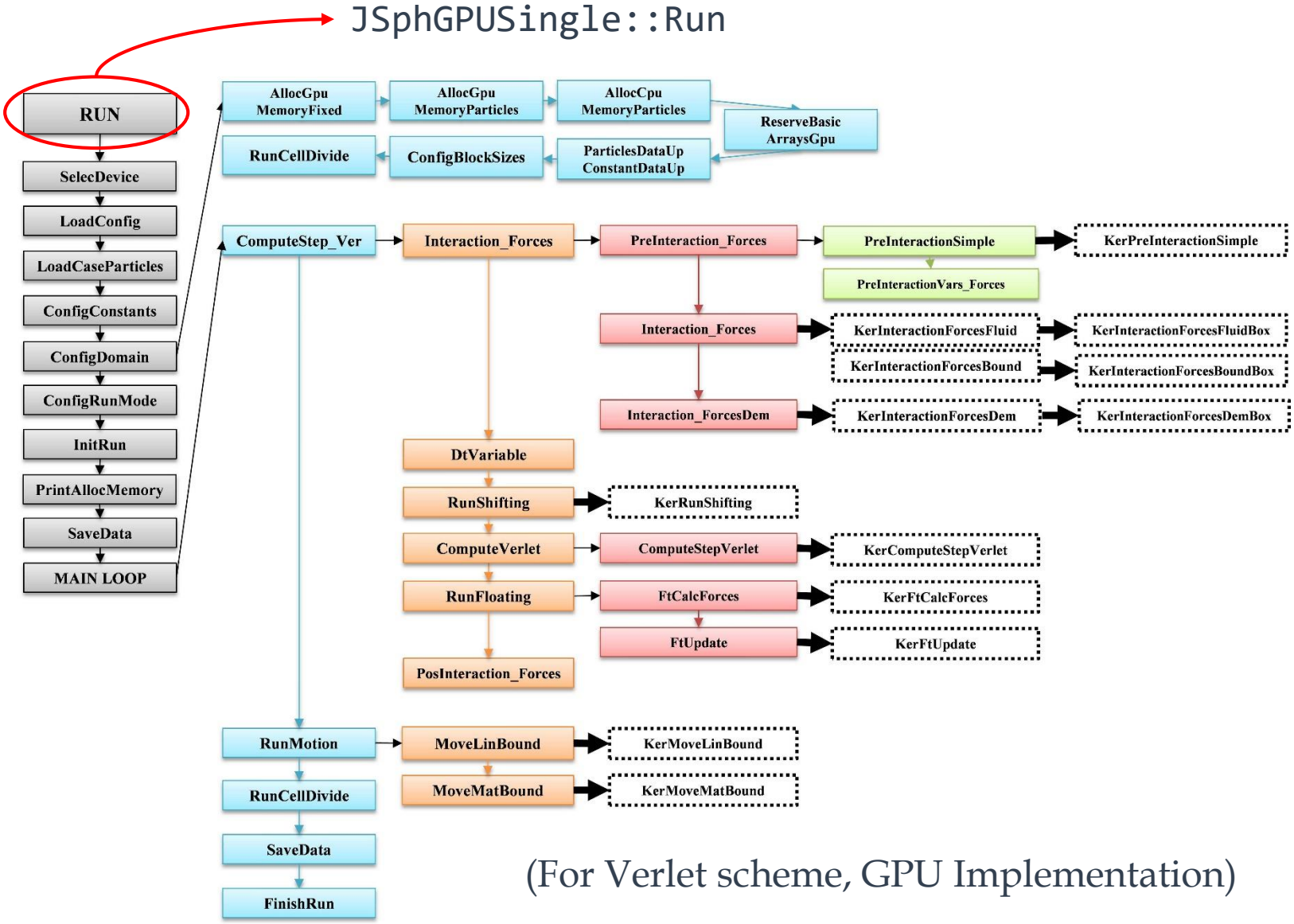
(Verlet scheme, CPU implementation)

DualSPHysics simulation run-flow



(Symplectic scheme, CPU implementation)

DualSPHysics simulation run-flow



(For Verlet scheme, GPU Implementation)

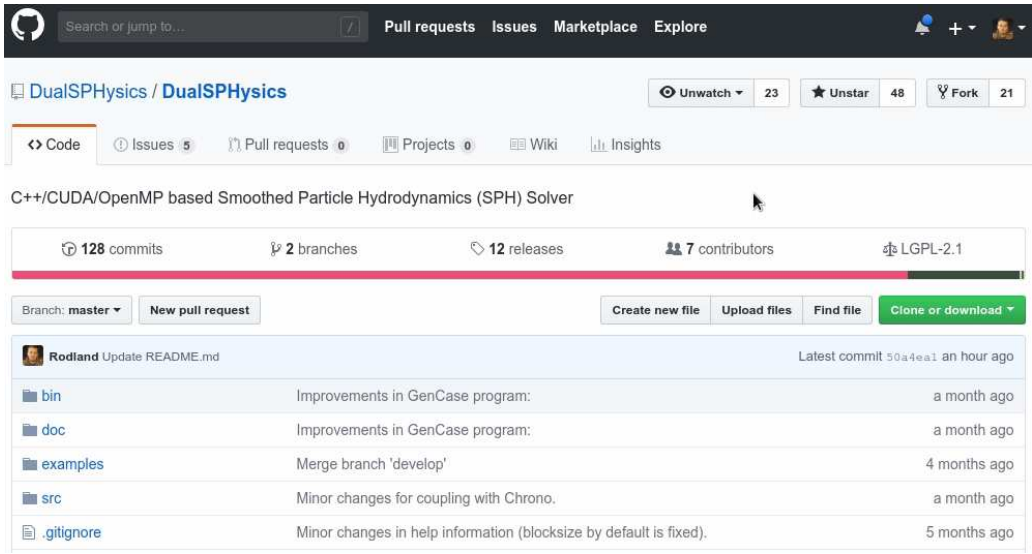
Let's start to work!

- Read the code carefully.
- Follow the execution path.
- Try to understand what the code does.
- Treat as a black box those parts don't have to modify.
- You don't need to understand all the code.
- The debugger is your friend.
- Remember to update all the references when you modify a function signature!!

Downloading the code

We are going to use the source code from GitHub repository

`https://github.com/DualSPHysics/DualSPHysics`



Windows:

Download git from:

https://git-scm.com/download/win

You can alternatively use a GUI like:

Sourcetree https://www.sourcetreeapp.com

https://tortoisegit.org


https://desktop.github.com

Linux:

Archlinux:

```
$ sudo pacman -S git
```

Debian/Ubuntu:

```
$ sudo apt-get install git
```

Fedora:

```
$ sudo dnf install git
```

Centos:

```
$ sudo yum install git
```

Clone the repository

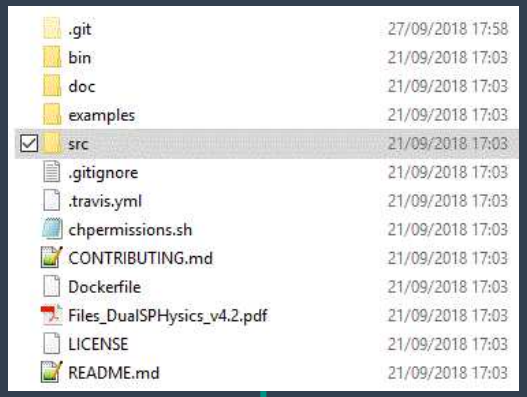
Windows: start **Git Bash**

Linux: launch a **terminal**

```
$ git clone https://github.com/DualSPHysics/DualSPHysics.git
Cloning into 'DualSPHysics'...
remote: Counting objects: 2402, done.
remote: Compressing objects: 100% (99/99), done.
remote: Total 2402 (delta 51), reused 57 (delta 25), pack-
reused 2278
Receiving objects: 100% (2402/2402), 291.16 MiB | 1.25 MiB/s,
done.
Resolving deltas: 100% (1668/1668), done.
Checking out files: 100% (292/292), done.
$ cd DualSPHysics
$ ls
CONTRIBUTING.md  Dockerfile  Files_DualSPHysics_v4.2.pdf
LICENSE  README.md  bin  chpermissions.sh  doc  examples  src
$
```

Compile the code on Windows:

You will need **Visual Studio 2015** and optionally **Nvidia CUDA**



Select the target:

- Release
- ReleaseCPU
- Debug
- DebugCPU

Press 'F7'

Compile the code on Linux:

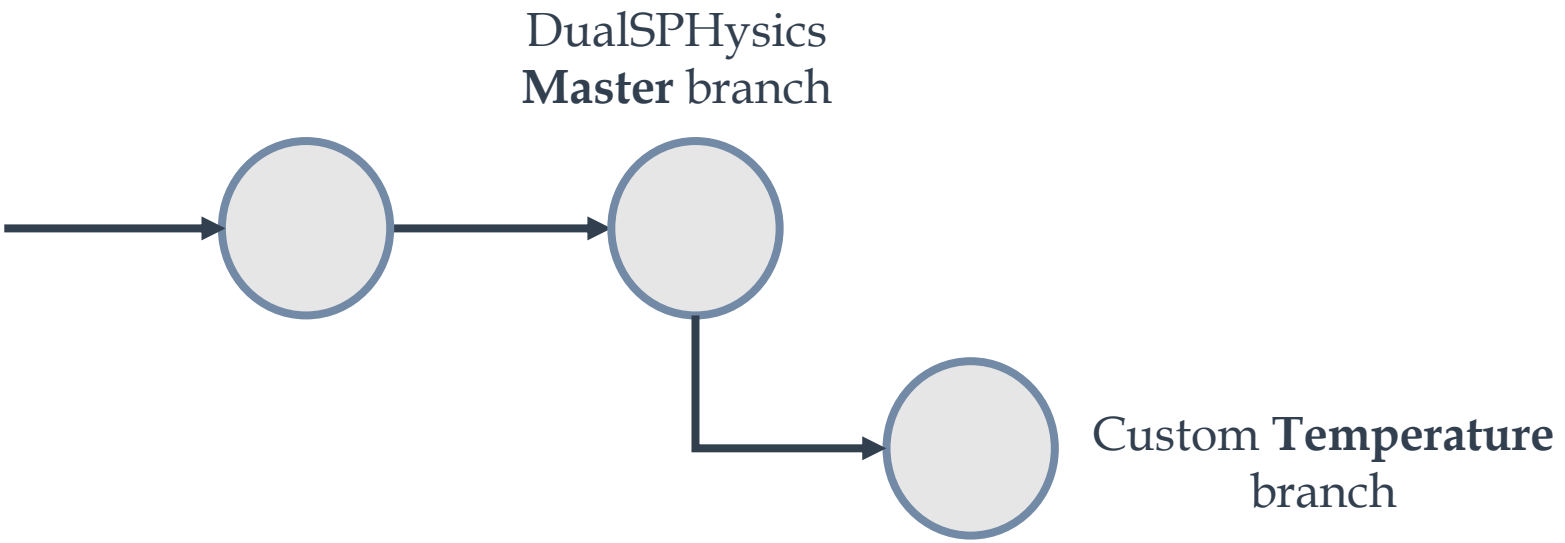
1. Go to the folder: `DualSPHysics/src/source/`
2. Edit Makefile and then:
   a. Set `DIRTOOLKIT` with the path to CUDA in your system e.g. `DIRTOOLKIT=/opt/cuda`
   b. If you want to use an specific G++ version: `CC=/usr/local/bin/g++-6`
3. Execute `make clean` to clean the environment.
4. Execute `make`

For CPU version: ignore CUDA and use the makefile `Makefile_cpu`.

To run make with this file:
`make -f Makefile_cpu`

You can alternatively use CMake. For more information check the wiki!
https://github.com/DualSPHysics/DualSPHysics/wiki

Create a GIT branch for your modifications




```
$ cd DualSPHysics
$ ls
CONTRIBUTING.md  Dockerfile  Files_DualSPHysics_v4.2.pdf  LICENSE
README.md  bin  chpermissions.sh  doc  examples  src
$ git checkout -b Temperature
Switched to a new branch 'Temperature'
```

The temperature ecuation

Energy conservation equation in its SPH form (Monaghan, 2005):

$$c_{p,a}\frac{\partial T}{\partial t}=\sum_b \frac{m_b}{\rho_a\rho_b}\frac{4k_ak_b}{k_a+k_b}(T_a-T_b)\frac{\nabla_a W_{ab}}{r_{ab}}$$

- c_p Specific heat capacity (constant)
- T Temperature
- m Mass (constant)
- ρ Density
- k Thermal conductivity (constant)
- W_{ab} Kernel function
- r_{ab} Distance

Define custom parameters in the XML file

```
<?xml version="1.0" encoding="UTF-8" ?>
<case>
. . .
<execution>
        <special>
            <temperature>
                <boundary mkbound="0">
                    <HeatKBound value="202.4" comment="Thermal conductivity of boundary particles
                      (default 54 for carbon steel at 293K)" units_comment="W/(m*K)" />
                    <HeatCpBound value="871" comment="Specific heat capacity of boundary particles
                      (default 465 for steel at 293K)" units_comment="J/(kg*K)" />
                    <HeatTempBound value="363" comment="Temperature of boundary particles"
                      units_comment="K"/>
                    <DensityBound value="2719" comment="Density of boundary particles"
                      units_comment="Kg/m^3"/>
                </boundary>
                <fluid>
                    <HeatKFluid value="0.6" comment="Thermal conductivity of fluid particles(default
                      0.6 for water at 293K)" units_comment="W/(m*K)" />
                    <HeatCpFluid value="4182" comment="Specific heat capacity of fluid particles
                      (default 4.18 for water 293K)" units_comment="J/(kg*K)" />
                    <HeatTempFluid value="293" comment="Temperature of fluid particles"
                      units_comment="K"/>
                </fluid>
            </temperature>
        </special>
. . .
</execution>
</case>
```

Define configuration variables to JSph.h file

JSph.h

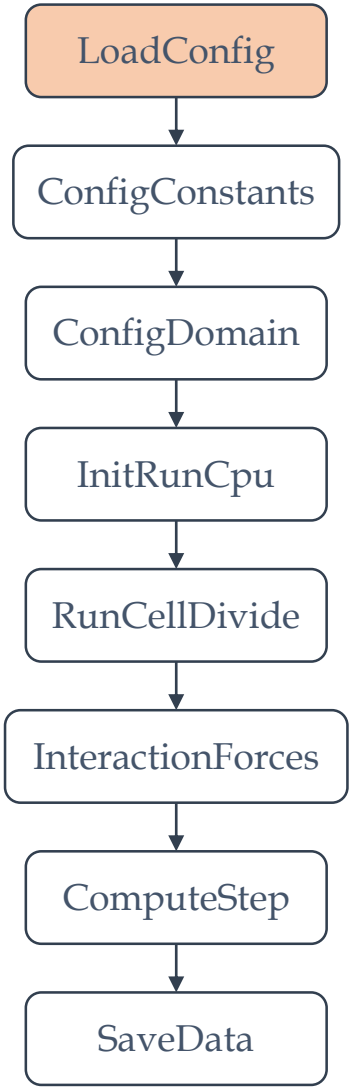
```
class JSph : protected JObject{
...
protected:
//=============================================
// Temperature configuration variables
//=============================================
bool HeatTransfer;     ///< Enable heat transfer
float HeatCpFluid;     ///< Specific heat capacity of fluid
particles
float HeatCpBound;     ///< Specific heat capacity of boundary
particles
float HeatKFluid;      ///< Thermal conductivity of fluid
particles
float HeatKBound;      ///< Thermal conductivity of boundary
particles
float HeatTempBound;   ///< Temperature of boundary particles K
float HeatTempFluid;   ///< Temperature of fluid particles K
float DensityBound;    ///< Density of boundary particles Kg/m^3
unsigned MkConstTempWall; ///< Mk of the constant temperature
wall boundary
//=============================================
```

JSph.cpp

```
void JSph::InitVars(){
. . .
//==============================
// Initialization of Temperature
//==============================
HeatTransfer = false;
HeatCpFluid = 0;
HeatCpBound = 0;
HeatKFluid = 0;
HeatKBound = 0;
HeatTempBound = 0;
HeatTempFluid = 0;
MkConstTempWall = 0;
DensityBound = 0;
//==============================
```

JSph class contains general configuration variables and functions needed for both CPU and GPU simulations.

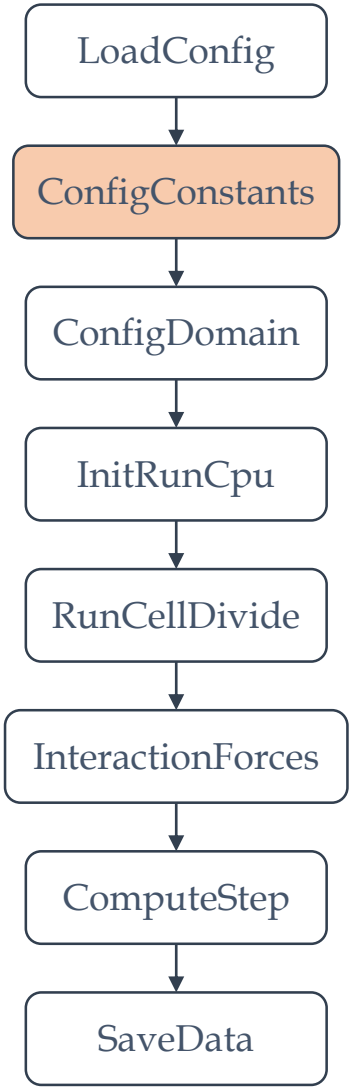
Load configuration parameters from the XML file



JSph.cpp

```
void JSph::LoadCaseConfig(){
. . .
//=====================================================
// Configuration of Temperature Parameters
//=====================================================
TiXmlNode* tempNode = xml.GetNode("case.execution.special.temperature", false);
if (tempNode) {
  HeatTransfer = true;
  MkConstTempWall = xml.ReadElementInt(tempNode->ToElement(), "boundary",
"mkbound");
  TiXmlNode* tempBoundNode =
xml.GetNode("case.execution.special.temperature.boundary", false);
  HeatCpBound = xml.ReadElementFloat(tempBoundNode, "HeatCpBound", "value");
  HeatKBound = xml.ReadElementFloat(tempBoundNode, "HeatKBound", "value");
  HeatTempBound = xml.ReadElementFloat(tempBoundNode, "HeatTempBound", "value");
  DensityBound = xml.ReadElementFloat(tempBoundNode, "DensityBound", "value");
  TiXmlNode* tempFluidNode =
xml.GetNode("case.execution.special.temperature.fluid", false);
  HeatCpFluid = xml.ReadElementFloat(tempFluidNode, "HeatCpFluid", "value");
  HeatKFluid = xml.ReadElementFloat(tempFluidNode, "HeatKFluid", "value");
  HeatTempFluid = xml.ReadElementFloat(tempFluidNode, "HeatTempFluid", "value");
}
//=====================================================
```

Log configuration parameters

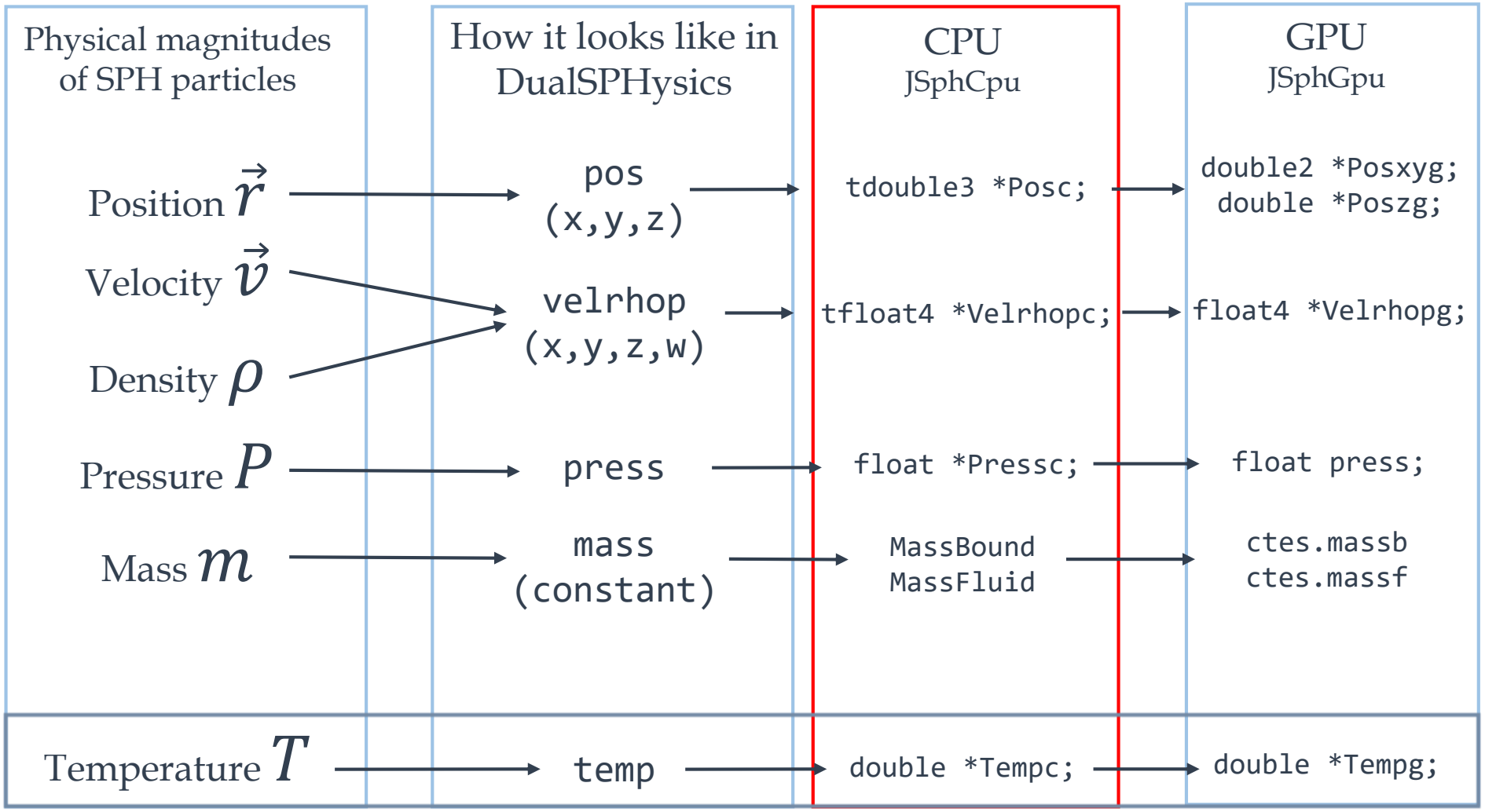


JSph.cpp

```
void JSph::VisuConfig()const{
. . .
//================================================
// Temperature: log configuration variables
//================================================
Log->Print(fun::VarStr("HeatTransfer", HeatTransfer));
if (HeatTransfer) {
   Log->Print(fun::VarStr("HeatCpFluid", HeatCpFluid));
   Log->Print(fun::VarStr("HeatCpBound", HeatCpBound));
   Log->Print(fun::VarStr("HeatKFluid", HeatKFluid));
   Log->Print(fun::VarStr("HeatKBound", HeatKBound));

   Log->Print(fun::VarStr("MkConstTempWall", MkConstTempWall));
   Log->Print(fun::VarStr("HeatTempBound", HeatTempBound));
   Log->Print(fun::VarStr("HeatTempFluid", HeatTempFluid));
   Log->Print(fun::VarStr("DensityBound", DensityBound));
}
//================================================
```

DualSPHysics variables nomenclature



Create computation arrays

JSphCpu.h

```
class JSphCpu : public JSph{
. . .
protected:
. . .
tfloat4 *Velrhopc;
double *Tempc; ///< Temperature: temperature for each particle.

. . .
tfloat4 *VelrhopM1c; ///<Verlet: in order to keep previous values.
double *TempM1c; ///< Temperature: temperature array to keep previous values for Verlet.

. . .
tfloat4 *VelrhopPrec;
double *TempPrec; /// Temperature: temperature array to keep previous values for Symplectic.

. . .
float *Arc;    ///< Density detivative.
float *Atempc; ///< Temperature derivative.
```

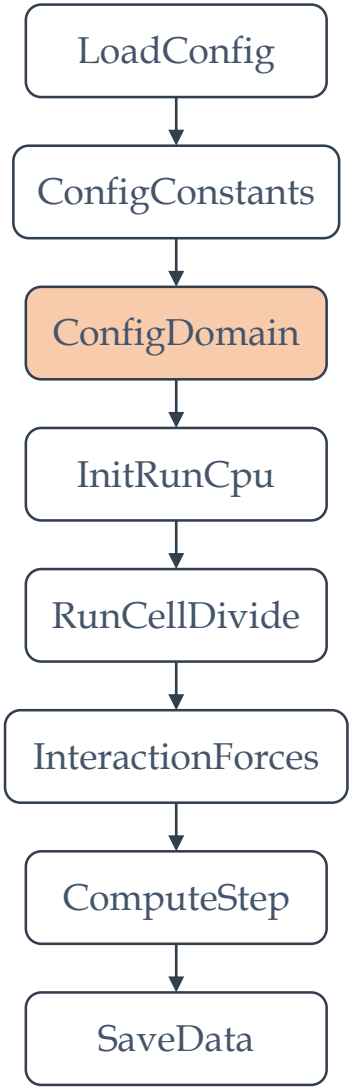
JSphCpu.cpp

```
void JSphCpu::InitVars(){
. . .

//==============================
// Initialization of Temperature computation vars
//==============================
Tempc = NULL;
TempM1c = NULL;
TempPrec = NULL;
Atempc = NULL;
//==============================
```

We are going to mimic what is done for density arrays!

Memory allocation for temperature vars



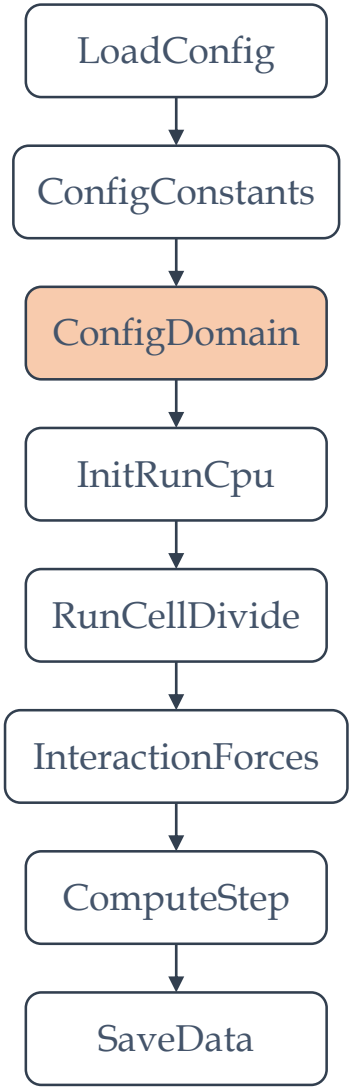
JSphCpu.cpp

```
void JSphCpu::AllocCpuMemoryParticles(unsigned np,float over){
  ArraysCpu->AddArrayCount(JArraysCpu::SIZE_4B,5); //-
idp,ar,viscdt,dcell,prrhop
  . . .
  ArraysCpu->AddArrayCount(JArraysCpu::SIZE_16B,1); //-velrhop

  //=============================================================
  // Temperature: AddArrayCount for Tempc & Atempc variable
  //=============================================================
  ArraysCpu->AddArrayCount(JArraysCpu::SIZE_8B, 2); // Tempc
  ArraysCpu->AddArrayCount(JArraysCpu::SIZE_4B, 1); // Atempc
  //=============================================================


  . . .
  if(TStep==STEP_Verlet){
    ArraysCpu->AddArrayCount(JArraysCpu::SIZE_16B,1); //-velrhopm1
    ArraysCpu->AddArrayCount(JArraysCpu::SIZE_8B, 1); // Temperature: TempM1
  }else if(TStep==STEP_Symplectic){
    . . .
    ArraysCpu->AddArrayCount(JArraysCpu::SIZE_16B,1); //-velrhoppre
    ArraysCpu->AddArrayCount(JArraysCpu::SIZE_8B, 1); // Temperature:
TempPrec
  }
```

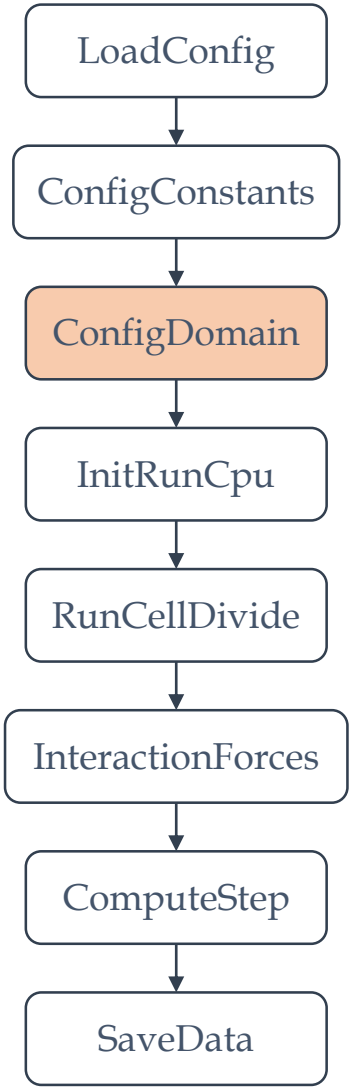
Memory allocation for temperature vars



JSphCpu.cpp

```
void JSphCpu::ReserveBasicArraysCpu(){
  . . .
  Velrhopc=ArraysCpu->ReserveFloat4();
  Tempc = ArraysCpu->ReserveDouble();
  if (TStep == STEP_Verlet) {
    VelrhopM1c = ArraysCpu->ReserveFloat4();
    TempM1c = ArraysCpu->ReserveDouble(); }
```

Initialize temperature arrays



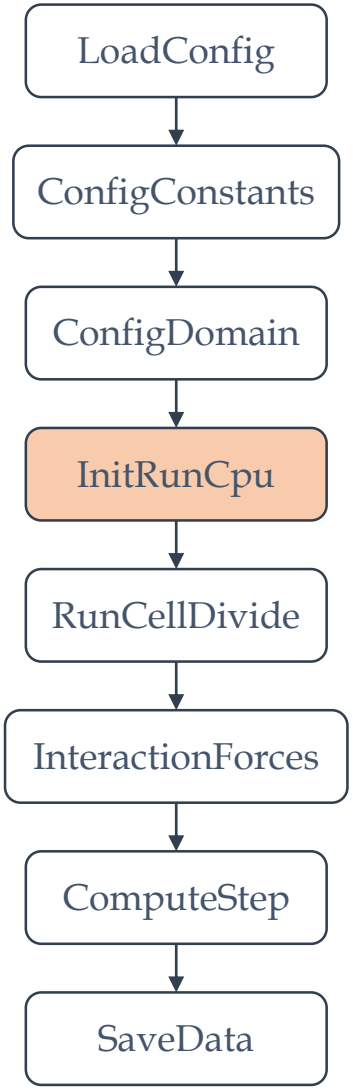
JSphCpuSingle.cpp

```
void JSphCpuSingle::ConfigDomain(){
. . .
//-Allocates memory in CPU for particles. | Reserva memoria en Cpu para
particulas.
AllocCpuMemoryParticles(Np,0);

//-Copies particle data.
ReserveBasicArraysCpu();
. . .

//==================================================
// Temperature: assign initial temperature
//==================================================
for (unsigned p = 0; p<Np; p++)
  Tempc[p] = HeatTempFluid;
for (unsigned c = 0; c<MkInfo->Size(); c++) {
  const JSphMkBlock* block = MkInfo->Mkblock(c);
  if (block->Mk == (MkConstTempWall + MkInfo->GetMkBoundFirst())) {
    for (unsigned p = block->Begin; p<block->Begin + block->Count; p++)
      Tempc[p] = HeatTempBound;
  }
}
//==================================================
```

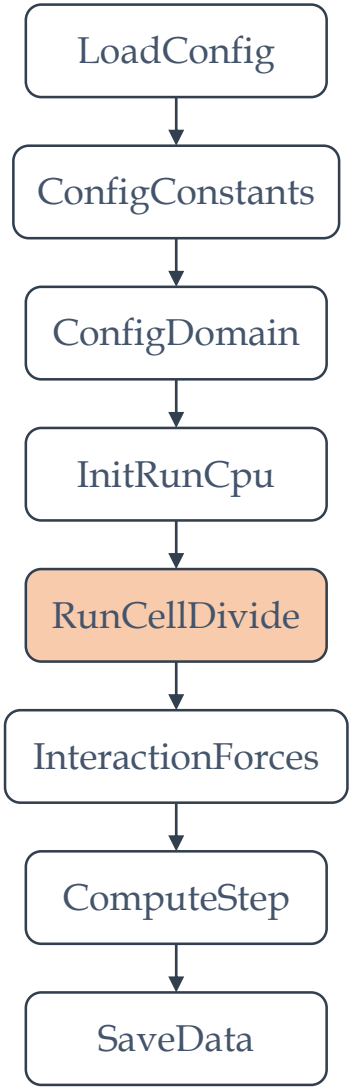
Initialize temperature arrays for Verlet



JSphCpu.cpp

```
void JSphCpu::InitRunCpu(){
InitRun();
if (TStep == STEP_Verlet) {
   memcpy(VelrhopM1c, Velrhopc, sizeof(tfloat4)*Np);
   memcpy(TempM1c, Tempc, sizeof(double)*Np); //
Temperature: Copy TempM1c and Tempc
}
. . .
```

Update the particle division in cells code



JSphCpuSingle.cpp

```
void JSphCpuSingle::RunCellDivide(bool updateperiodic){
  . . .
  CellDivSingle->SortArray(Velrhopc);
  CellDivSingle->SortArray(Tempc); // Overloaded function: sort
Tempc (double) array.
  if(TStep==STEP_Verlet){
    CellDivSingle->SortArray(VelrhopM1c);
    CellDivSingle->SortArray(TempM1c);
  }else if(TStep==STEP_Symplectic && (PosPrec || VelrhopPrec)){
    . . .
    CellDivSingle->SortArray(VelrhopPrec);
    CellDivSingle->SortArray(TempPrec);
  }
  . . .
}
```

Update the particle division in cells code

JSphCpu.cpp

```
unsigned JSphCpu::GetParticlesData(unsigned n,unsigned pini,bool cellorderdecode,
bool onlynormal,unsigned *idp,tdouble3 *pos,tfloat3 *vel,float *rhop,
double *temp,typecode *code){
. . .
if(temp)memcpy(temp, Tempc + pini, sizeof(double)*n); // Temperature: copy values
from Tempc.
//-Eliminate non-normal particles (periodic & others).
if(onlynormal){
  . . .
  unsigned ndel=0;
  for(unsigned p=0;p<n;p++){
  bool normal=CODE_IsNormal(code2[p]);
  if(ndel && normal){
    const unsigned pdel=p-ndel;
    . . .
    rhop[pdel] =rhop[p];
    temp[pdel] = temp[p]; // Temperature
    . . .
  }
. . .
```

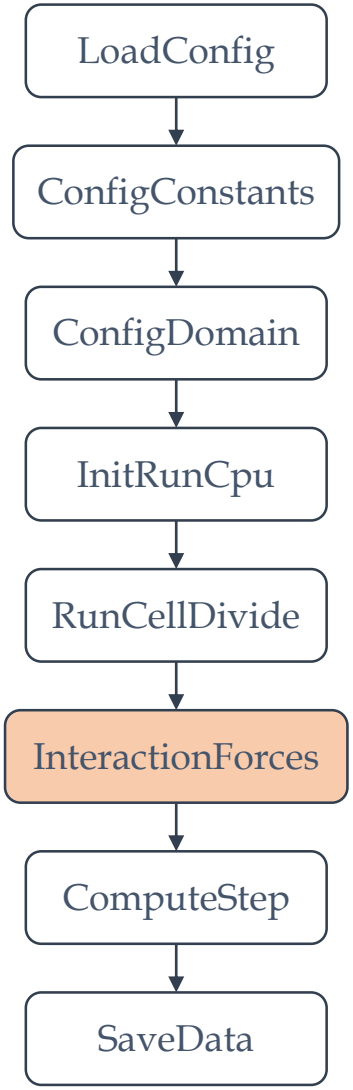
The temperature ecuation

Energy conservation equation in its SPH form (Monaghan, 2005):

$$c_{p,a}\frac{\partial T}{\partial t}=\sum_b \frac{m_b}{\rho_a\rho_b}\frac{4k_ak_b}{k_a+k_b}(T_a-T_b)\frac{\nabla_a W_{ab}}{r_{ab}}$$

- c_p Specific heat capacity (constant)
- T Temperature
- m Mass (constant)
- ρ Density
- k Thermal conductivity (constant)
- W_{ab} Kernel function
- r_{ab} Distance

Initialize temperature derivative array



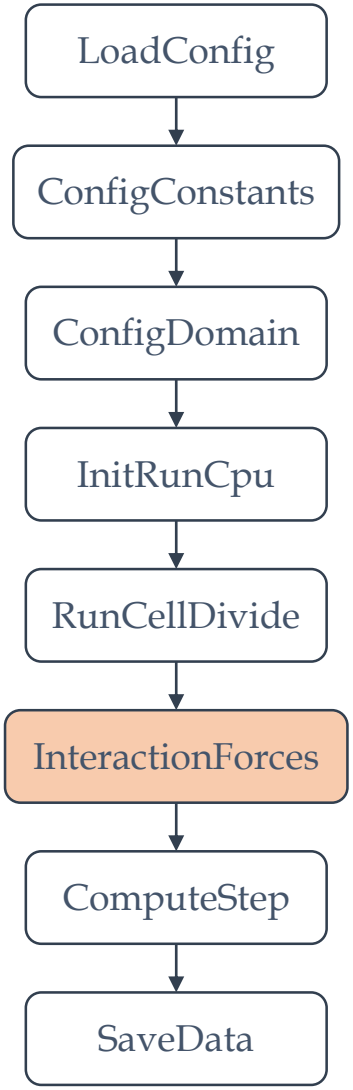
JSphCpu.cpp

```
void JSphCpu::PreInteraction_Forces(TpInter tinter){
. . .
//-Assign memory.
Arc=ArraysCpu->ReserveFloat();
Acec=ArraysCpu->ReserveFloat3();
Atempc=ArraysCpu->ReserveFloat(); // Temperature: reserve memory for Atempc
```

JSphCpu.cpp

```
void JSphCpu::PreInteractionVars_Forces(TpInter tinter,unsigned np,unsigned
npb){
//-Initialize Arrays.
const unsigned npf=np-npb;
memset(Arc,0,sizeof(float)*np); //Arc[]=0
. . .
memset(Atempc, 0, sizeof(float)*np); //Temperature: Atempc[]=0
```

Compute temperature derivative (Fluid)

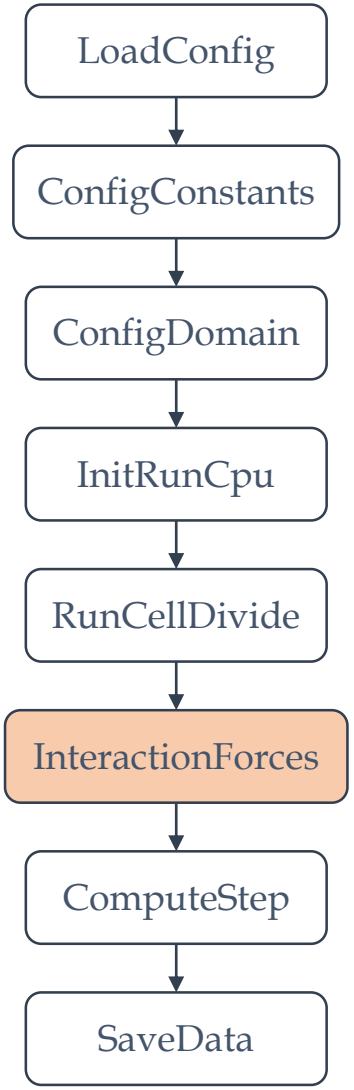


JSphCpu.cpp

```
void JSphCpu::InteractionForcesFluid (. . ., const tfloat4 *velrhop,const
double *temp,. . .,float *ar,tfloat3 *ace,float *atemp,. . .)const{
. . .
for(int p1=int(pinit);p1<pfin;p1++){
  float visc=0,arp1=0,deltap1=0;
  float atempp1 = 0.; // Temperature: declare local variable.
. . .
//-Obtain data of particle p1.
. . .
const float rhopp1=velrhop[p1].w;
const double tempp1 = temp[p1]; // Temperature: load temperature of particle
p1.
. . .
//-Cubic Spline, Wendland or Gaussian kernel.
float frx,fry,frz,fabc;
if(tker==KERNEL_Wendland)GetKernelWendland(rr2,drx,dry,drz,frx,fry,frz,fabc);
else
if(tker==KERNEL_Gaussian)GetKernelGaussian(rr2,drx,dry,drz,frx,fry,frz,fabc);
else if(tker==KERNEL_Cubic)GetKernelCubic(rr2,drx,dry,drz,frx,fry,frz,fabc);
. . .
```

Remember to update all the references when you modify a function signature!!

Kernels



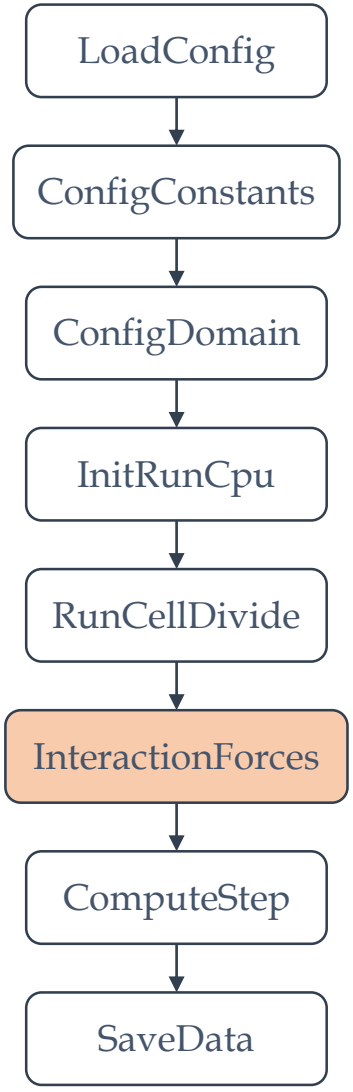
JSphCpu.cpp

```
void JSphCpu::GetKernelWendland(float rr2,float drx,float dry,float drz
,float &frx,float &fry,float &frz,float &fabc)const {
  const float rad=sqrt(rr2);
  const float qq=rad/H;
  //-Wendland kernel.
  const float wqq1=1.f-0.5f*qq;
  const float fac=Bwen*qq*wqq1*wqq1*wqq1/rad;
  frx=fac*drx; fry=fac*dry; frz=fac*drz;
  fabc=fac;
}

void JSphCpu::GetKernelGaussian(float rr2,float drx,float dry,float drz
,float &frx,float &fry,float &frz,float &fabc)const {
  . . .
  frx=fac*drx; fry=fac*dry; frz=fac*drz;
  fabc=fac;
}

void JSphCpu::GetKernelCubic(float rr2,float drx,float dry,float drz
,float &frx,float &fry,float &frz,float &fabc)const {
  . . .
  frx=fac*drx; fry=fac*dry; frz=fac*drz;
  fabc=fac;
}
```

Compute temperature derivative (Fluid)



$$c_{p,a}\frac{\partial T}{\partial t} = \sum_b \frac{m_b}{\rho_a \rho_b}\frac{4k_a k_b}{k_a + k_b}(T_a - T_b)\frac{\nabla_a W_{ab}}{r_{ab}}$$

JSphCpu.cpp

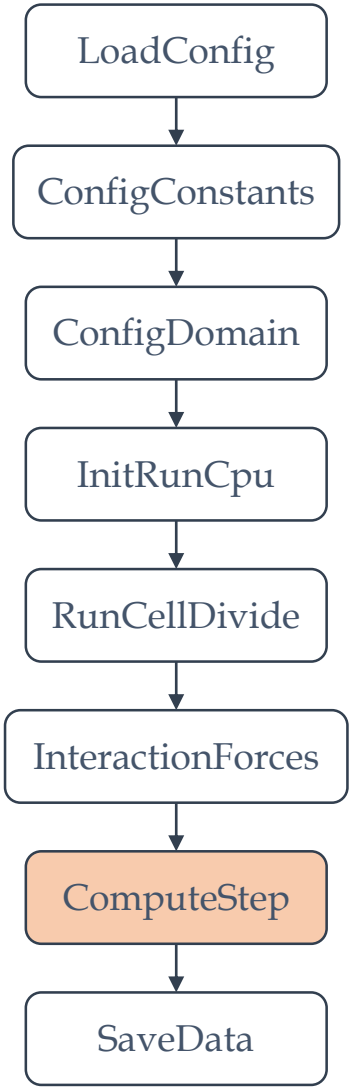
```
. . .
//-Density derivative.
const float dvx=velp1.x-velrhop[p2].x, dvy=velp1.y-velrhop[p2].y, dvz=velp1.z-
velrhop[p2].z;
if(compute)arp1+=massp2*(dvx*frx+dvy*fry+dvz*frz);
. . .
// Temperature: compute temperature derivative
if (compute) {
  float heatKp2 = (boundp2 ? HeatKBound : HeatKFluid); // Check if p2 is bound or
fluid then assign the respective thermal conductivity K.
  float rhopp2 = (boundp2 ? DensityBound : velrhop[p2].w); // Check if p2 is bound
or fluid and assign the respective density.
  const double dtemp = tempp1 - temp[p2]; // (dtemp=tempp1-tempp2)
  const float tempConst = (4 * massp2*HeatKFluid*heatKp2) /
(HeatCpFluid*rhopp1*rhopp2*(HeatKFluid + heatKp2));
  atempp1 += float(tempConst*dtemp*fabc);
}
. . .
//-Sum results together. | Almacena resultados.
if(shift||arp1||acep1.x||acep1.y||acep1.z||visc){
  . . .
  ar[p1]+=arp1;
  atemp[p1] += atempp1; // Temperature: Add atemp for particle p1.
```

Compute temperature derivative (Boundaries)

JSphCpu.cpp

```
template<bool psingle,TpKernel tker,TpFtMode ftmode> void JSphCpu::InteractionForcesBound (unsigned
n,unsigned pinit, tint4 nc,int hdiv,unsigned cellinitial,const unsigned *beginendcell,tint3
cellzero,const unsigned *dcell,const tdouble3 *pos,const tfloat3 *pspos,const tfloat4 *velrhop,const
double *temp,const typecode *code,const unsigned *idp,float &viscdt,float *ar, float *atemp)const {
. . .
for(int p1=int(pinit);p1<pfin;p1++){
  float visc=0,arp1=0;
  float atempp1 = 0; // Temperature: initialize temp derivative to 0 for particle p1.
//-Load data of particle p1. | Carga datos de particula p1.
. . .
   const double tempp1 = temp[p1]; // Temperature: load p1 temperature.
   const float rhopp1 = DensityBound; // Temperature: load density for boundary.
. . .
//-Search for neighbours in adjacent cells. | Busqueda de vecinos en celdas adyacentes.
for(int z=zini;z<zfin;z++){
. . .
if(compute){
//-Density derivative.
const float dvx=velp1.x-velrhop[p2].x, dvy=velp1.y-velrhop[p2].y, dvz=velp1.z-velrhop[p2].z;
if(compute)arp1+=massp2*(dvx*frx+dvy*fry+dvz*frz);
       // Temperature: compute temperature derivative
       //================================================
       const double dtemp = tempp1 - temp[p2]; // Temperature: (dtemp=tempp1-tempp2)
       const float tempConst = (4 * massp2*HeatKFluid*HeatKBound) /
(HeatCpBound*rhopp1*velrhop[p2].w*(HeatKFluid + HeatKBound));
       atempp1 += float(tempConst*dtemp*fabc);
       //================================================
. . .
```

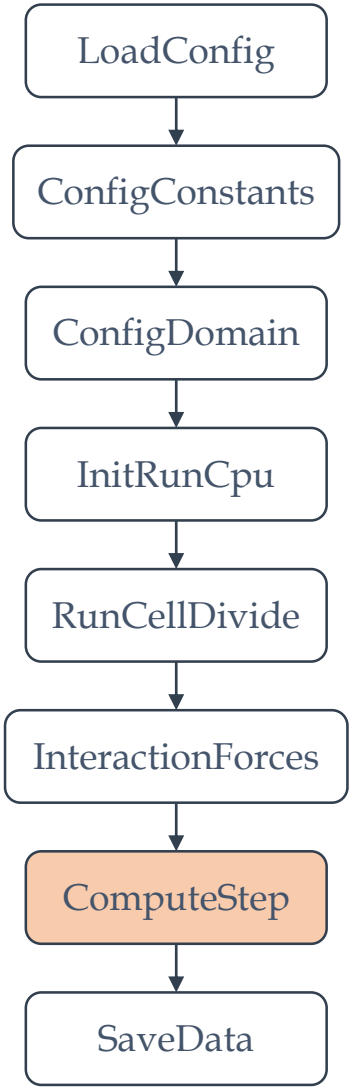
Compute Verlet



JSphCpu.cpp

```
void JSphCpu::ComputeVerlet(double dt){
  . . .
  ComputeVerletVarsFluid<false> (Velrhopc,Velrhopc,Tempc,
dt,dt,Posc, Dcellc,Codec,VelrhopM1c,TempM1c);
  ComputeVelrhopBound(Velrhopc,Tempc,dt,VelrhopM1c,TempM1c);
  . . .
  //-New values are calculated en VelrhopM1c.
  swap(Velrhopc,VelrhopM1c); //-Swap Velrhopc & VelrhopM1c.
  swap(Tempc, TempM1c); // Temperature: swap Tempc & TempM1c
  . . .
}
```

Compute Verlet



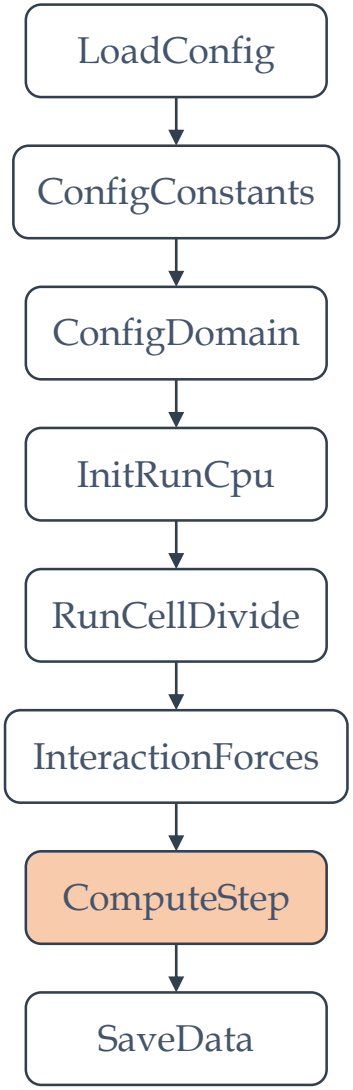
JSphCpu.cpp

```
void JSphCpu::ComputeVelrhopBound(const tfloat4* velrhopold, const double*
tempold,double armul,tfloat4* velrhopnew,double* tempnew)const{
  . . .
  for(int p=0;p<npb;p++){
    const float rhopnew=float(double(velrhopold[p].w)+armul*Arc[p]);
    velrhopnew[p]=TFloat4(0,0,0,(rhopnew<RhopZero? RhopZero: rhopnew));
    tempnew[p]=tempold[p]; // Temperature: constant temperature on
boundaries for this implementation.
  }
}
```

JSphCpu.cpp

```
template<bool shift> void JSphCpu::ComputeVerletVarsFluid(const tfloat4
*velrhop1, const tfloat4 *velrhop2,const double *tempp2,double dt,double
dt2,tdouble3 *pos, unsigned *dcell,typecode *code,tfloat4
*velrhopnew,double *tempnew)const {
  . . .
  for(int p=pini;p<pfin;p++){
    //-Calculate density. | Calcula densidad.
    const float rhopnew=float(double(velrhop2[p].w)+dt2*Arc[p]);
    tempnew[p] = tempp2[p]+dt2*Atempc[p]; // Temperature: compute new
temperature
  . . .
```

Free memory



JSphCpu.cpp

```
void JSphCpu::PosInteraction_Forces(){
//-Free memory assigned in PreInteraction_Forces().
ArraysCpu->Free(Arc); Arc=NULL;
ArraysCpu->Free(Acec); Acec=NULL;
ArraysCpu->Free(Deltac); Deltac=NULL;
ArraysCpu->Free(ShiftPosc); ShiftPosc=NULL;
ArraysCpu->Free(ShiftDetectc); ShiftDetectc=NULL;
ArraysCpu->Free(Pressc); Pressc=NULL;
ArraysCpu->Free(PsPosc); PsPosc=NULL;
ArraysCpu->Free(SpsGradvelc); SpsGradvelc=NULL;

ArraysCpu->Free(Atempc); Atempc = NULL; // Temperature:
free Atempc and reset the pointer to NULL
}
```

Compute symplectic (predictor)

JSphCpu.cpp

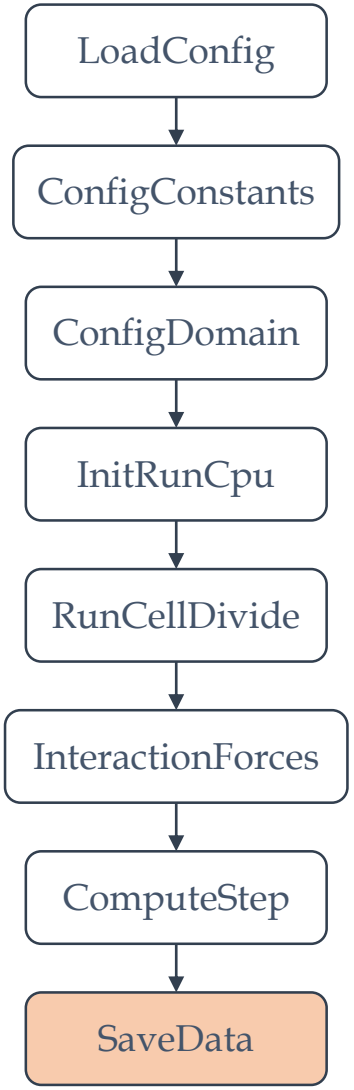
```
template<bool shift> void JSphCpu::ComputeSymplecticPreT(double dt){
  . . .
  VelrhopPrec=ArraysCpu->ReserveFloat4();
  TempPrec = ArraysCpu->ReserveDouble(); // Temperature: reserve memory
  //-Change data to variables Pre to calculate new data.
  . . .
  swap(VelrhopPrec,Velrhopc); //Put value of Velrhop[] in VelrhopPre[].
  swap(TempPrec,Tempc); // Temperature
  . . .
  for(int p=0;p<npb;p++){
    const tfloat4 vr=VelrhopPrec[p];
    const float rhopnew=float(double(vr.w)+dt05*Arc[p]);
    Velrhopc[p]=TFloat4(vr.x,vr.y,vr.z,(rhopnew<RhopZero? RhopZero: rhopnew));//-Avoid
fluid particles being absorbed by boundary ones.
    Tempc[p]=TempPrec[p]; //Temperature: constant temperature for the boundary
  }
  . . .
  for(int p=npb;p<np;p++){
    const float rhopnew=float(double(VelrhopPrec[p].w)+dt05*Arc[p]);
    Tempc[p]=TempPrec[p]+dt05*Atempc[p]; //Temperature: Calculate new temperature for
the fluid
. . .
```

Compute symplectic (corrector)

JSphCpu.cpp

```
template<bool shift> void JSphCpu::ComputeSymplecticCorrT(double dt){
  . . .
  for(int p=npb;p<np;p++){
    const double epsilon_rdot=(-double(Arc[p])/double(Velrhopc[p].w))*dt;
    const float rhopnew=float(double(VelrhopPrec[p].w) * (2.-epsilon_rdot)
/(2.+epsilon_rdot));
    //=================================================
    // Temperature
    const double epsilon_tdot=(-double(Atempc[p])/Tempc[p])*dt;
    Tempc[p]=TempPrec[p]*(2.-epsilon_tdot)/(2.+epsilon_tdot);
    //=================================================
. . .
//-Free memory assigned to variables Pre and ComputeSymplecticPre().
ArraysCpu->Free(PosPrec); PosPrec=NULL;
ArraysCpu->Free(VelrhopPrec); VelrhopPrec=NULL;
ArraysCpu->Free(TempPrec); TempPrec= NULL; // Temperature: free memory
```

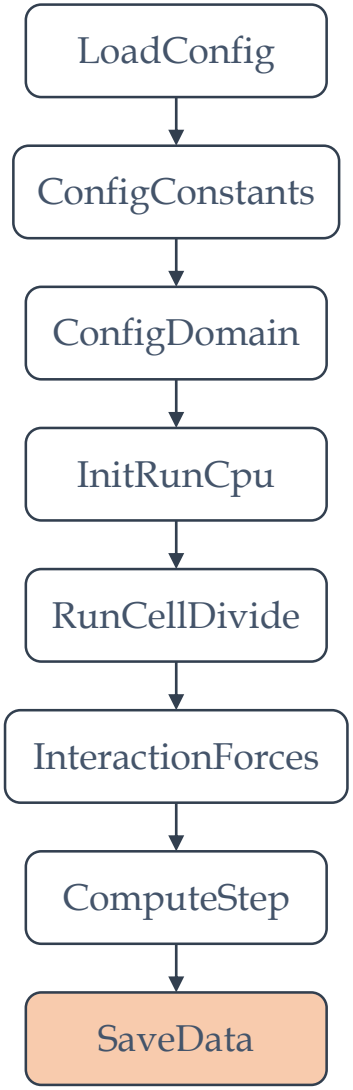
Save temperature to "bi4" output files



JSphCpuSingle.cpp

```
void JSphCpuSingle::SaveData(){
  . . .
  float *rhop=NULL;
  double *temp = NULL; // Temperature: temporal array.
  if(save){
    //-Assign memory and collect particle values.
    . . .
    rhop=ArraysCpu->ReserveFloat();
    temp = ArraysCpu->ReserveDouble(); // Temperature: allocate
memory.
    unsigned npnormal=
GetParticlesData(Np,0,true,PeriActive!=0,idp,pos,vel,rhop,temp,N
ULL); // Temperature: new temp param
    . . .
  }
  . . .
  JSph::SaveData(npsave,idp,pos,vel,rhop,temp,1,vdom,&infoplus);
// Temperature: new temp param
  . . .
  ArraysCpu->Free(rhop);
  ArraysCpu->Free(temp); // Temperature: free memory.
```

Save temperature to "bi4" output files



JSph.cpp

```
void JSph::SaveData(unsigned npok,const unsigned *idp,const tdouble3
*pos,const tfloat3 *vel,const float *rhop,const double *temp,unsigned
ndom,const tdouble3 *vdom,const StInfoPartPlus *infoplus){
. . .
//-Stores data files of particles.
SavePartData(npok,nout,idp,pos,vel,rhop,temp,ndom,vdom,infoplus);
. . .
```

JSph.cpp

```
void JSph::SavePartData(unsigned npok,unsigned nout,const unsigned
*idp,const tdouble3 *pos,const tfloat3 *vel,const float *rhop,const double
*temp,unsigned ndom,const tdouble3 *vdom,const StInfoPartPlus *infoplus){
  //-Stores particle data and/or information in bi4 format.
  if(DataBi4){
    . . .
    if(SvData&SDAT_Binx){
      . . .
       if (temp)DataBi4->AddPartData("Temp", npok, temp);
// Temperature: add temperature data.
```

Using new variable in post-processing tools

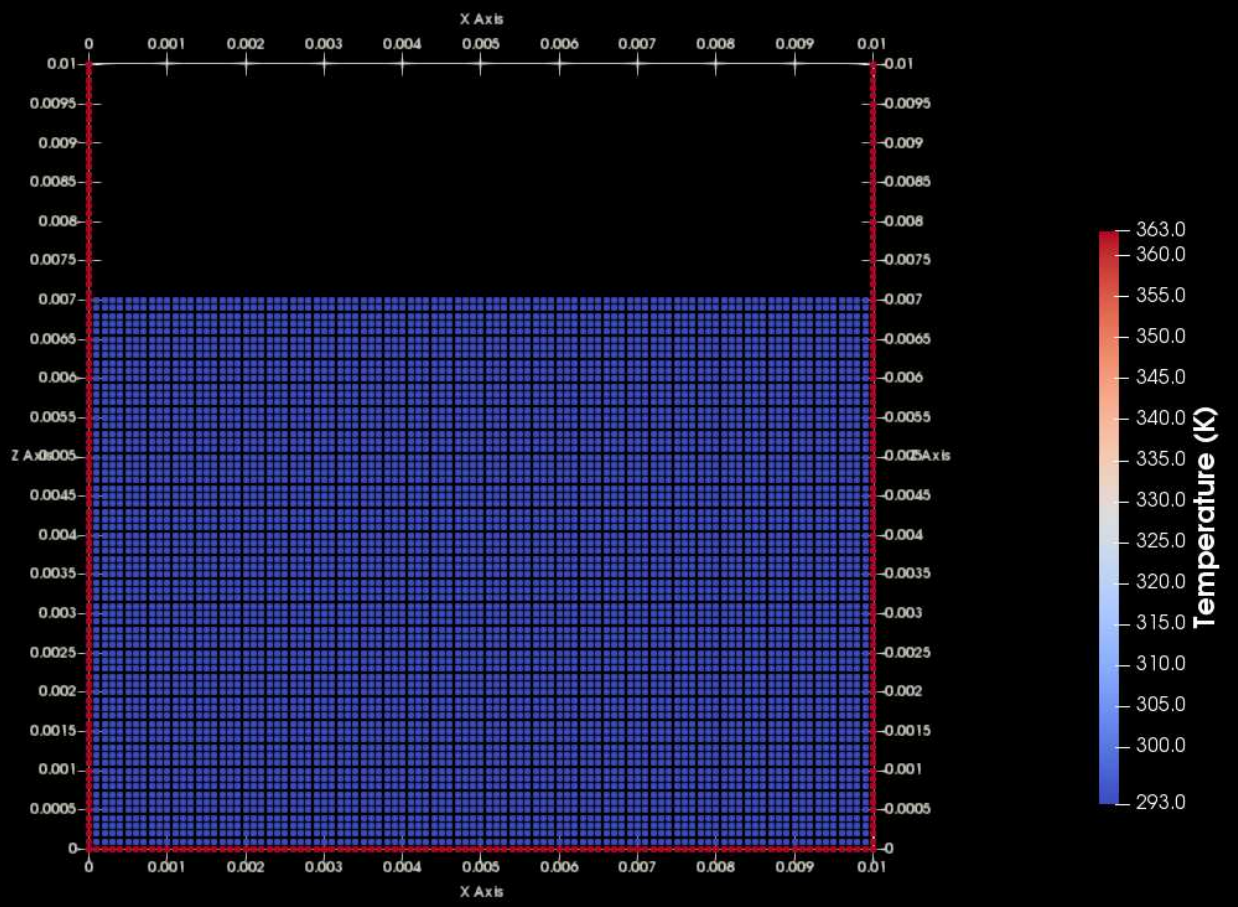
```
-vars:+temp
```

```
> PartVTK4_win64.exe -dirin diroutdata -savevtk Particles -vars:+temp
```

```
> MeasureTool4_win64.exe -dirin diroutdata -points PointsVelocity.txt
-onlytype:-all,+fluid -savevtk PointsVelocity
-vars:-all,+vel.x,+vel.m,+temp
```

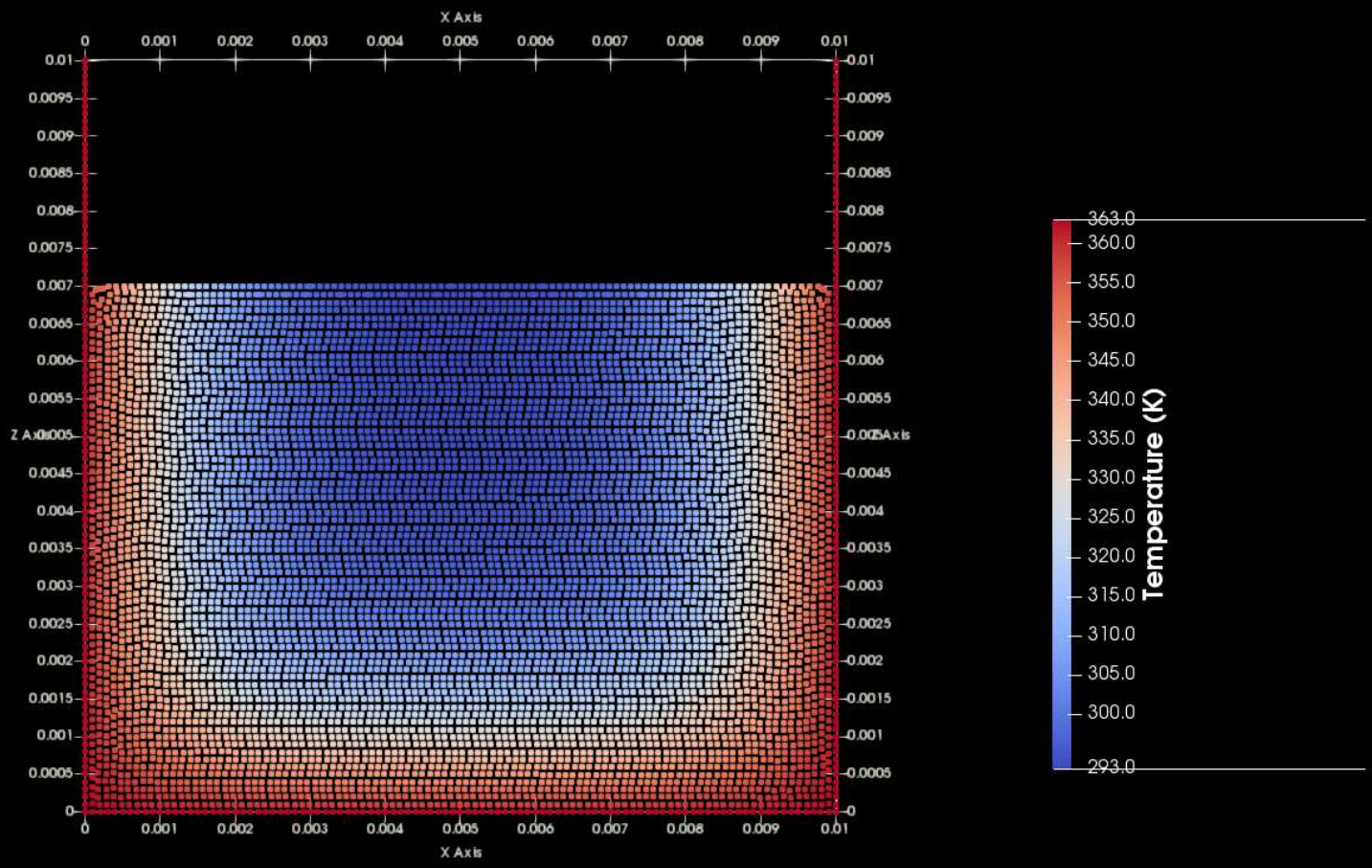
Testing the implementation

Time: 0.00 s

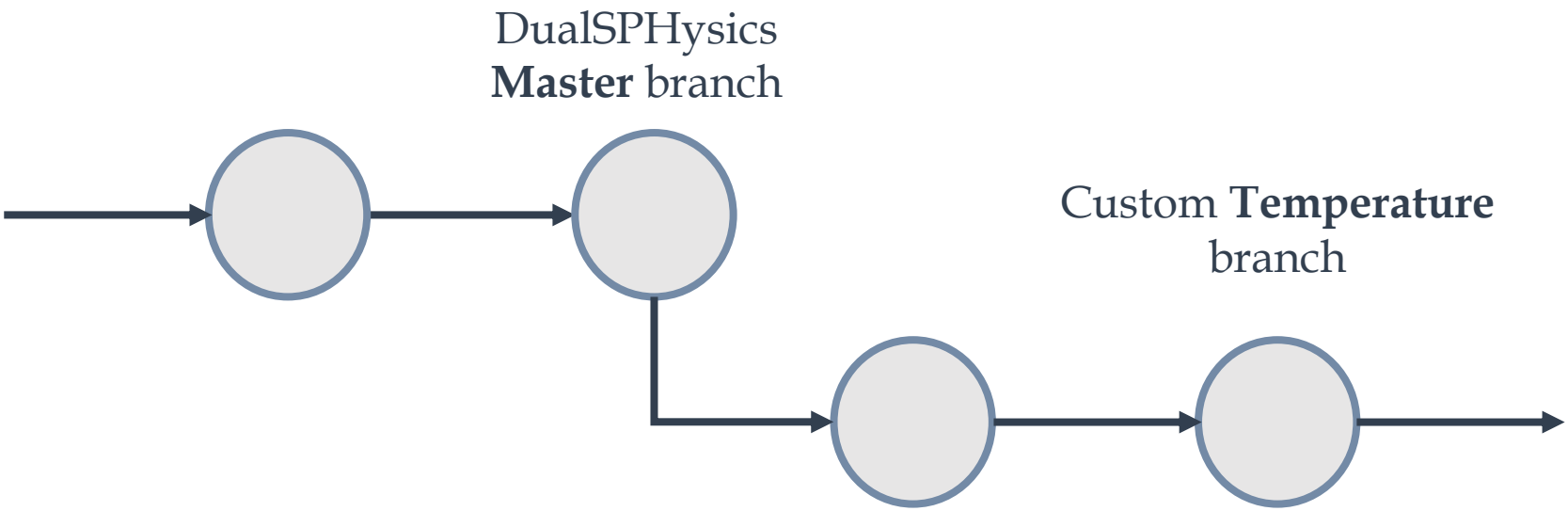


Testing the implementation

Time: 10.00 s



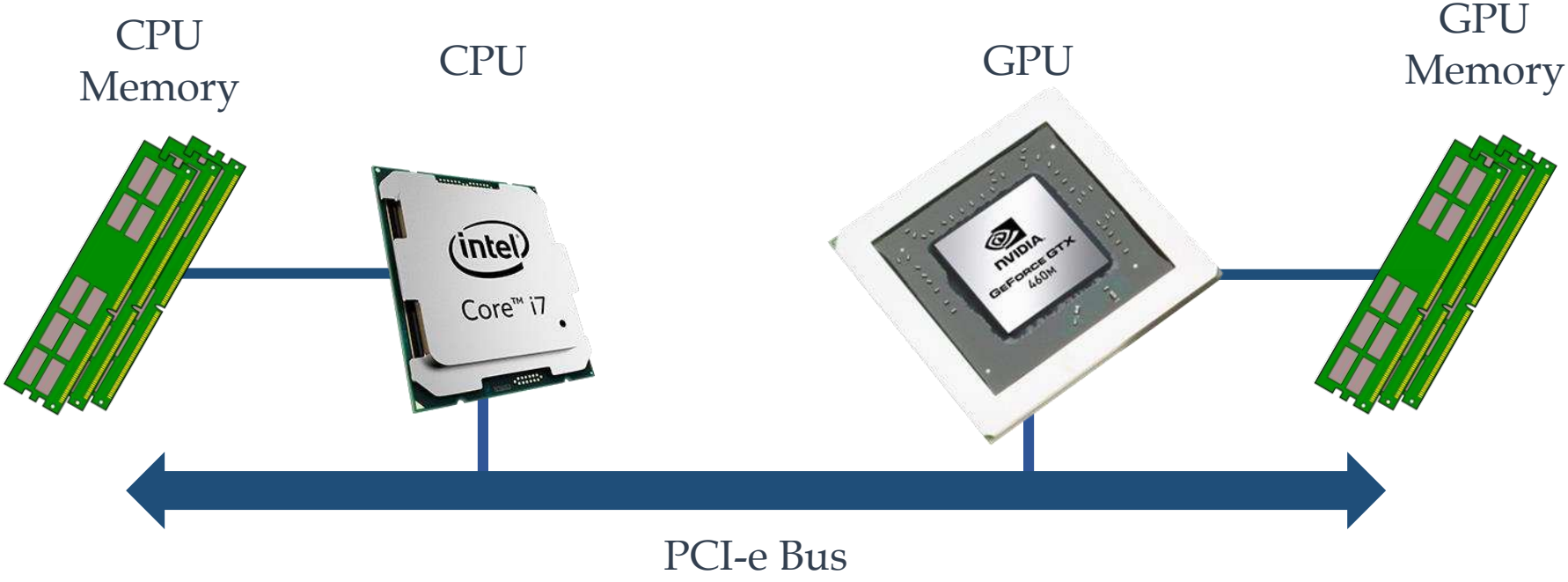
Commit your changes to GIT!




```
$ cd DualSPHysics
$ ls
CONTRIBUTING.md  Dockerfile  Files_DualSPHysics_v4.2.pdf  LICENSE
README.md  bin  chpermissions.sh  doc  examples  src
$ git commit -am “Temperature implemented on CPU.”
[Temperature 32d1e91] Temperature implemented on CPU.
 9 files changed, 292 insertions(+), 517 deletions(-)
```

The GPU implementation!

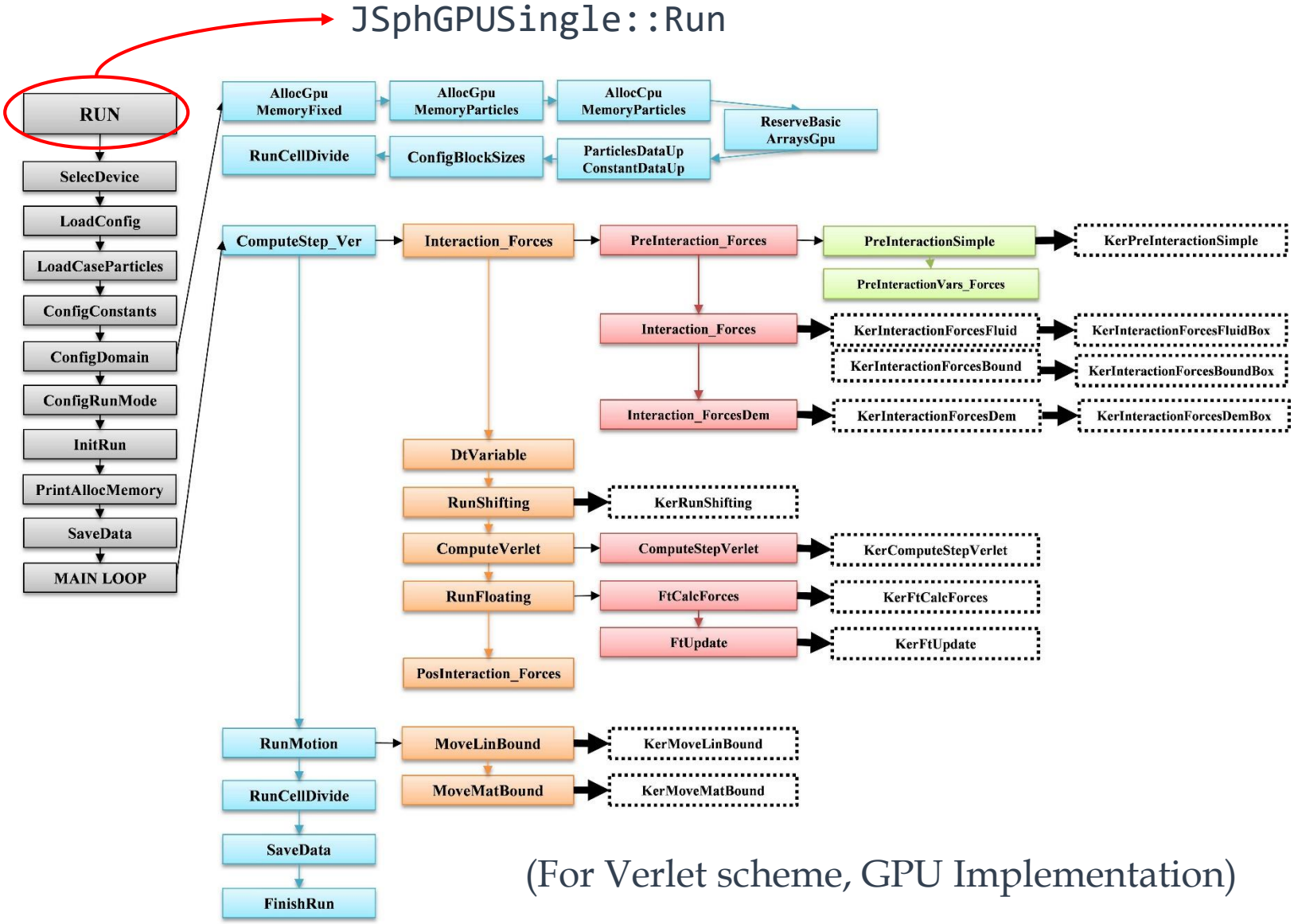
- GPUs are programmed in CUDA
- CUDA is similar to C/C++
- GPU Programming has some considerations



Source files that we will modify

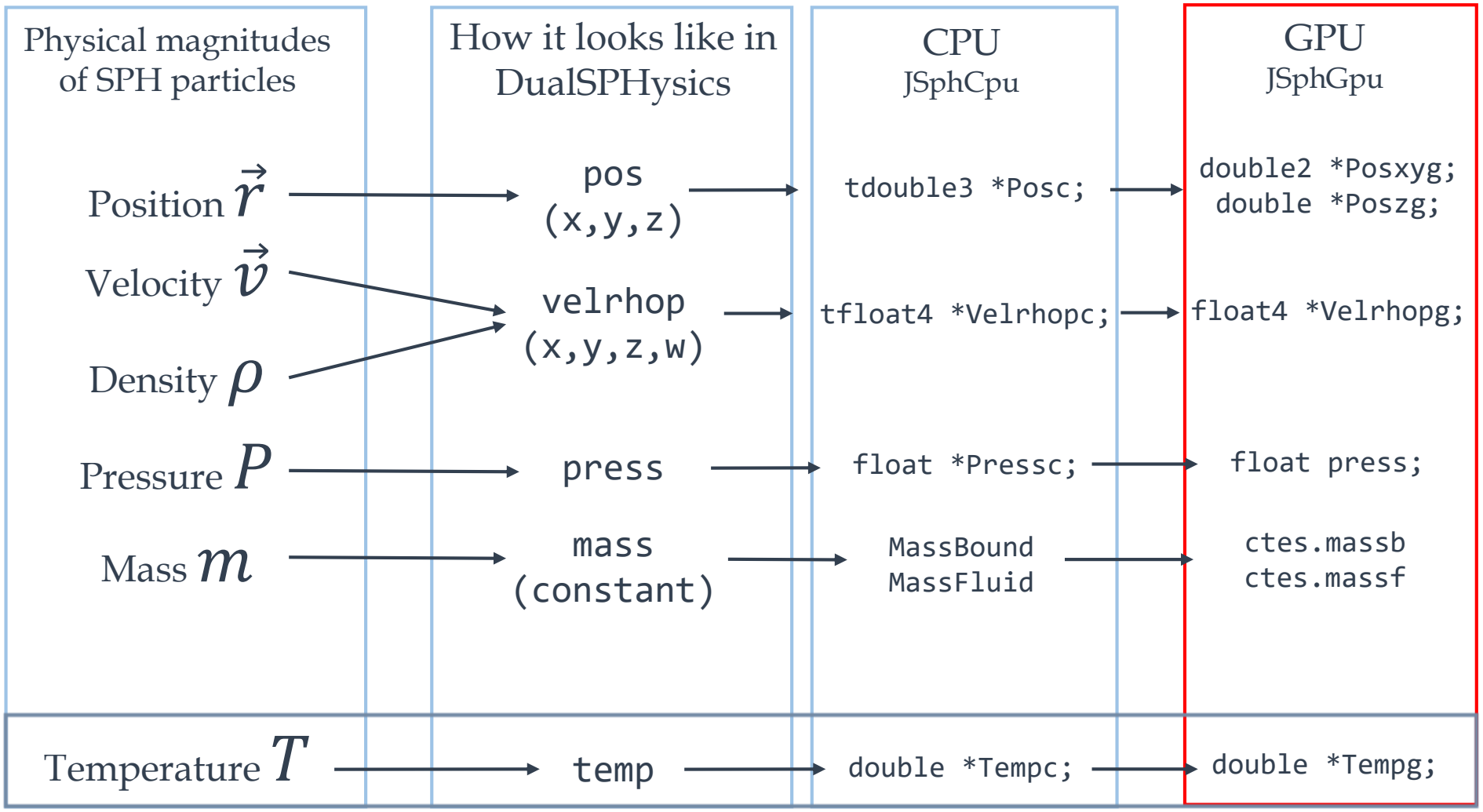
Common	`JSph.h/cpp`	Attributes and functions shared by CPU & GPU simulations.
CPU	`JSphCpu.h/cpp`	Attributes and functions used only by CPU simulations.
	`JSphCpuSingle.h/cpp`	Attributes and functions used to arrange CPU simulations.
	`JCellDivCpu.h/cpp`	Generates neighbor list for CPU simulations.
GPU	`JSphGpu.h/cpp`	Attributes and functions used only by GPU simulations.
	`JSphGpu_ker.h/cu`	CUDA kernels for GPU simulations.
	`JSphGpuSingle.h/cpp`	Attributes and functions used to arrange GPU simulations.
	`JCellDivGpu.h/cpp`	Generates neighbor list for GPU simulations.
	`JCellDivGpu_ker.h/cu`	CUDA Kernels for GPU neighbor generation.

DualSPHysics simulation run-flow



(For Verlet scheme, GPU Implementation)

DualSPHysics variables nomenclature



Define constants for GPU

JSphGpu_ker.h

```
typedef struct{
  . . .

  // Temperature: constants for temperature
  float
    HeatCpFluid,
    HeatCpBound,
    HeatKFluid,
    HeatKBound,
    DensityBound;

} StCteInteraction;
```

Create computation arrays

JSphGpu.cpp

```
class JSphGpu : public JSph{
  . . .
  //-Variables holding particle data for the execution
  tfloat4 *Velrhop;
  double *Temp;
  . . .
  //-Auxiliary variables for the conversión
  float *AuxRhop;
  double *AuxTemp;
  . . .
  //-Variables holding particle data for the execution
  float4 *Velrhopg;
  double *Tempg;
  //-Variables for compute step: VERLET.
  float4 *VelrhopM1g;
  double *TempM1g;
  //-Variables for compute step: SYMPLECTIC.
  . . .
  float4 *VelrhopPreg;
  double *TempPreg;
  . . .
  float *Arg;
  float *Atempg;
```

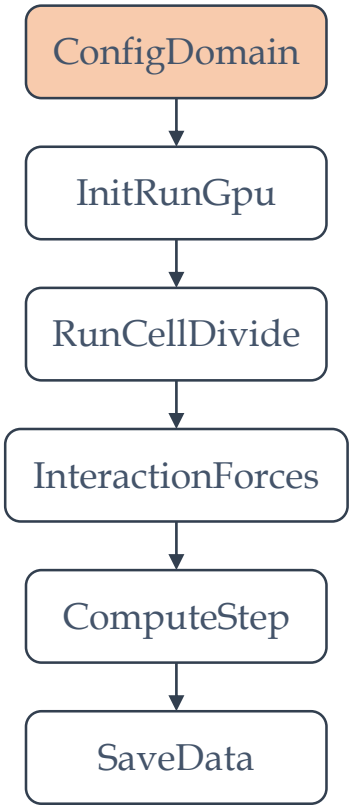
JSphGpu.cpp

```
JSphGpu::JSphGpu(bool withmpi):
JSph(false,withmpi){
. . .
Velrhop=NULL;
Temp = NULL;
. . .
AuxRhop=NULL;
AuxTemp = NULL;
. . .
}
```

JSphGpu.cpp

```
void JSphGpu::InitVars(){
. . .
Tempg = NULL;
TempM1g = NULL;
TempPreg = NULL;
Atempg = NULL;
```

Initialize temperature arrays

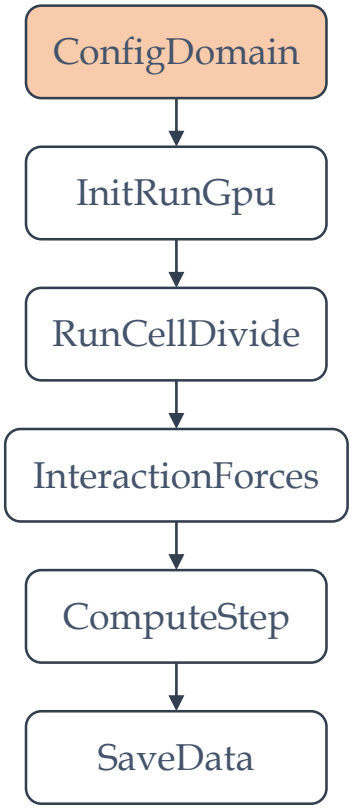


JSphGpuSingle.cpp

```
void JSphGpuSingle::ConfigDomain(){
  . . .
  //-Allocates GPU memory for particles.
  . . .
  AllocGpuMemoryParticles(Np,0);
  //-Allocates memory on the CPU.
  . . .
  AllocCpuMemoryParticles(Np);
  //-Copies particle data.
  . . .
  memcpy(Velrhop,PartsLoaded->GetVelRhop(),sizeof(tfloat4)*Np);

  //==================================================
  // Temperature: assign initial temperature
  for (unsigned p = 0; p<Np; p++) Temp[p] = HeatTempFluid;
  for (unsigned c = 0; c<MkInfo->Size(); c++) {
       const JSphMkBlock* block = MkInfo->Mkblock(c);
       if (block->Mk == (MkConstTempWall + MkInfo->GetMkBoundFirst())) {
           for (unsigned p = block->Begin; p<block->Begin + block->Count; p++){
               Temp[p] = HeatTempBound;
           }
       }
  }
  //==================================================
```

Memory management for computation arrays



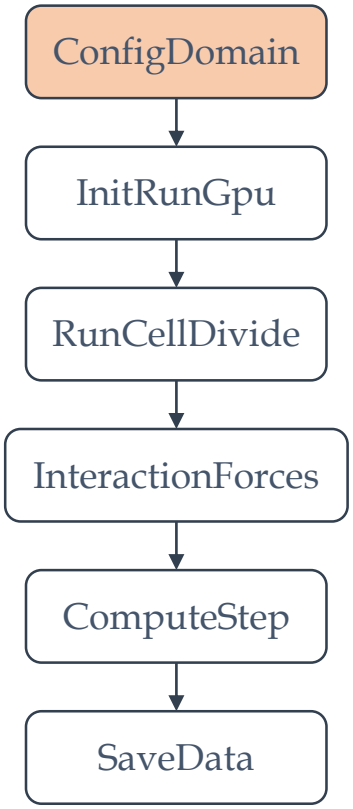
JSphGpu.cpp

```
void JSphGpu::AllocGpuMemoryParticles(unsigned np,float over){
. . .
ArraysGpu->AddArrayCount(JArraysGpu::SIZE_4B,4); //-idp,ar,viscdt,dcell
. . .
ArraysGpu->AddArrayCount(JArraysGpu::SIZE_16B,4); //-velrhop,posxy

ArraysGpu->AddArrayCount(JArraysGpu::SIZE_8B, 2); // Tempg
ArraysGpu->AddArrayCount(JArraysGpu::SIZE_4B, 1); // Atempg

if(TStep==STEP_Verlet){
  ArraysGpu->AddArrayCount(JArraysGpu::SIZE_16B,1); //-velrhopm1
  ArraysGpu->AddArrayCount(JArraysGpu::SIZE_8B, 1); // Temperature: TempM1
}
else if(TStep==STEP_Symplectic){
  . . .
  ArraysGpu->AddArrayCount(JArraysGpu::SIZE_16B,2); //-posxypre,velrhoppre
  ArraysGpu->AddArrayCount(JArraysGpu::SIZE_8B, 1); // Temperature: TempPrec
}
. . .
```

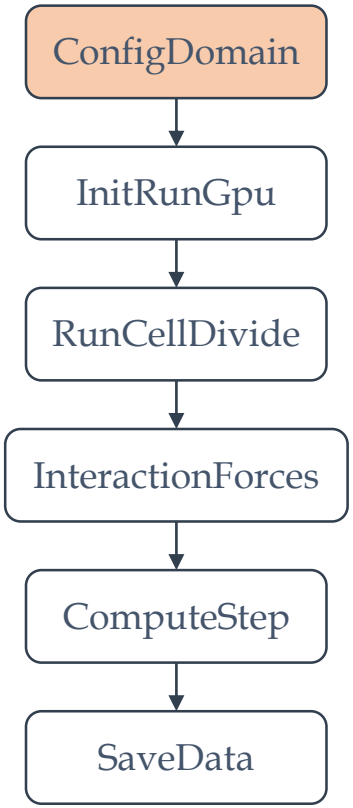
Memory management for computation arrays



JSphGpu.cpp

```
void JSphGpu::ReserveBasicArraysGpu(){
  . . .
  Velrhopg=ArraysGpu->ReserveFloat4();
  Tempg= ArraysGpu->ReserveDouble();
  if (TStep == STEP_Verlet) {
    VelrhopM1g = ArraysGpu->ReserveFloat4();
    TempM1g = ArraysGpu->ReserveDouble();
  }
  . . .
```

Memory transfer to GPU



JSphGpu.cpp

```
void JSphGpu::ParticlesDataUp(unsigned n){
  . . .

cudaMemcpy(Velrhopg,Velrhop,sizeof(float4)*n,cudaMemcpyHostToDevice);
cudaMemcpy(Tempg ,Temp ,sizeof(double)*n ,cudaMemcpyHostToDevice);
  . . .
}
```

JSphGpu.cpp

```
void JSphGpu::ConstantDataUp(){
  StCteInteraction ctes;
  memset(&ctes,0,sizeof(StCteInteraction));
  . . .
  //=====================================
  // Temperature: copy constants to GPU
  //=====================================
  ctes.HeatCpFluid = HeatCpFluid;
  ctes.HeatCpBound = HeatCpBound;
  ctes.HeatKFluid = HeatKFluid;
  ctes.HeatKBound = HeatKBound;
  ctes.DensityBound = DensityBound;
  //=====================================
  . . .
}
```

Memory management for CPU arrays

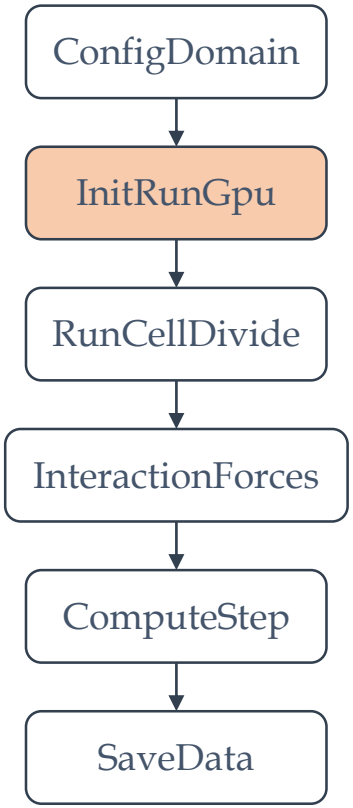
JSphGpu.cpp

```
void JSphGpu::AllocCpuMemoryParticles(unsigned np){
  . . .
  if(np>0){
    try{
      . . .
      Velrhop=new tfloat4[np];  MemCpuParticles+=sizeof(tfloat4)*np;
      . . .
      AuxRhop=new float[np];    MemCpuParticles+=sizeof(float)*np;
      Temp = new double[np];    MemCpuParticles += sizeof(double)*np;
      AuxTemp = new double[np]; MemCpuParticles += sizeof(double)*np;
      . . .
```

JSphGpu.cpp

```
void JSphGpu::FreeCpuMemoryParticles(){
  . . .
  delete[] Velrhop; Velrhop=NULL;
  delete[] Temp; Temp = NULL;
  . . .
  delete[] AuxRhop; AuxRhop=NULL;
  delete[] AuxTemp; AuxTemp = NULL;
}
```

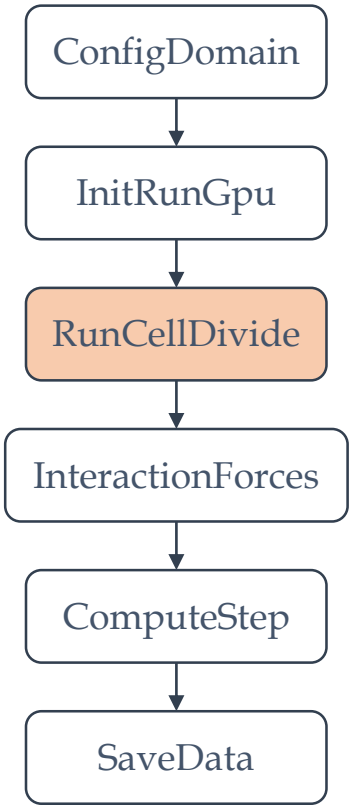
Initialize temperature arrays for Verlet



JSphGpu.cpp

```
void JSphGpu::InitRunGpu(){
  InitRun();
  if (TStep == STEP_Verlet) {
    cudaMemcpy(VelrhopM1g, Velrhopg, sizeof(float4)*Np,
cudaMemcpyDeviceToDevice);
    cudaMemcpy(TempM1g, Tempg, sizeof(double)*Np, cudaMemcpyDeviceToDevice);
  }
  if(TVisco==VISCO_LaminarSPS)cudaMemset(SpsTaug,0,sizeof(tsymatrix3f)*Np);
  if(CaseNfloat)InitFloating();
  CheckCudaError("InitRunGpu","Failed initializing variables for
execution.");
}
```

Update the particle division in cells code



JSphGpuSingle.cpp

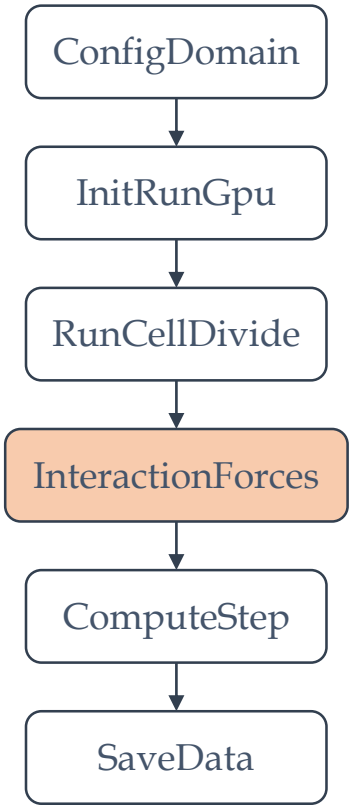
```
void JSphGpuSingle::RunCellDivide(bool updateperiodic){
  . . .
  float4* velrhopg=ArraysGpu->ReserveFloat4();
  CellDivSingle->SortBasicArrays (Idpg,Codeg,Dcellg,Posxyg,Poszg,Velrhopg,idpg,
codeg,dcellg,posxyg,poszg,velrhopg);
  double* tempg = ArraysGpu->ReserveDouble();
  CellDivSingle->SortDataArrays(Tempg, tempg);
  . . .
  swap(Velrhopg,velrhopg); ArraysGpu->Free(velrhopg);
  swap(Tempg, tempg); ArraysGpu->Free(tempg);
  . . .
  if(TStep==STEP_Verlet){
    float4* velrhopg=ArraysGpu->ReserveFloat4();
    CellDivSingle->SortDataArrays(VelrhopM1g,velrhopg);
    swap(VelrhopM1g,velrhopg); ArraysGpu->Free(velrhopg);
    //================================================
    // Temperature: sort array TempM1g
    CellDivSingle->SortDataArrays(TempM1g, tempg); // Overloaded function!
    swap(TempM1g, tempg);
    //================================================
  }
  ArraysGpu->Free(tempg);
  . . .
```

Update the particle division in cells code

JSphGpuSingle.cpp

```
. . .
else if(TStep==STEP_Symplectic && (PosxyPreg || PoszPreg || VelrhopPreg)){
  . . .
  float4* velrhopg=ArraysGpu->ReserveFloat4();
  CellDivSingle->SortDataArrays(PosxyPreg,PoszPreg,VelrhopPreg,posxyg,poszg,velrhopg);
  . . .
  swap(VelrhopPreg,velrhopg); ArraysGpu->Free(velrhopg);
  //================================================
  // Temperature: sort array TempPreg
  double* tempg = ArraysGpu->ReserveDouble();
  CellDivSingle->SortDataArrays(TempPreg, tempg);
  swap(TempPreg, tempg);
  ArraysGpu->Free(tempg);
  //================================================
}
. . .
```

Initialize derivative array



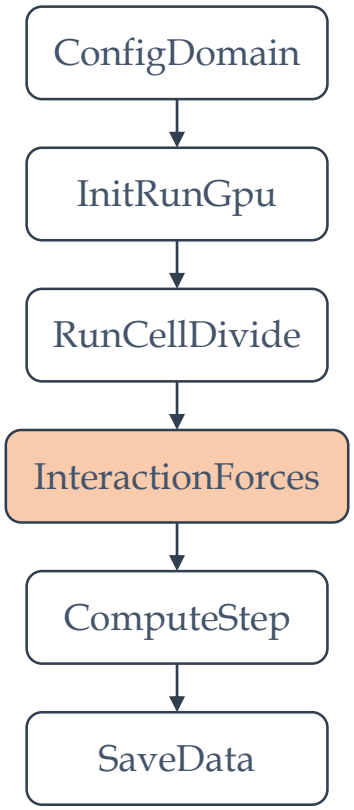
JSphGpu.cpp

```
void JSphGpu::PreInteraction_Forces(TpInter tinter){
  . . .
  //-Allocates memory.
  . . .
  Arg=ArraysGpu->ReserveFloat();
  Atempg=ArraysGpu->ReserveFloat();
  . . .
```

JSphGpu.cpp

```
void JSphGpu::PreInteractionVars_Forces(TpInter tinter,unsigned
np,unsigned npb){
  . . .
  cudaMemset(Arg,0,sizeof(float)*np);
  cudaMemset(Atempg, 0, sizeof(float)*np);
  . . .
}
```

Update interaction forces functions

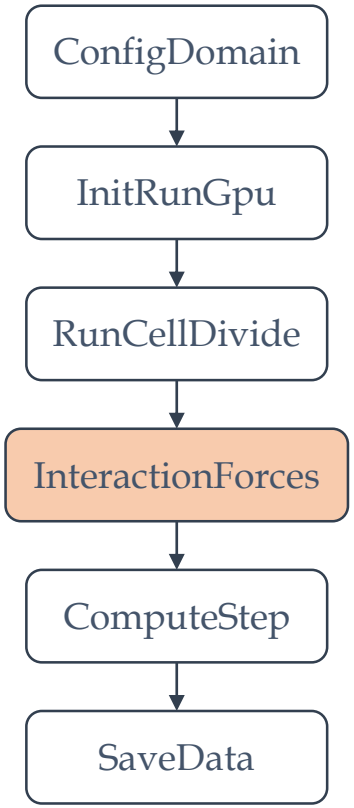


JSphGpu_ker.cu

```
__global__ void KerInteractionForcesFluid (. . .,const float4 *velrhop,const double
*temp,. . .,float *ar,float *atemp, . . .){
  unsigned p=blockIdx.y*gridDim.x*blockDim.x + blockIdx.x*blockDim.x + threadIdx.x;
// Number of particle.
  if(p<n){
    unsigned p1=p+pinit; //-Number of particle.
    float visc=0,arp1=0,deltap1=0;
    float atempp1 = 0.;
    . . .
    //-Obtains basic data of particle p1.
    . . .
    double tempp1 = temp[p1];
    . . .
    if(pfin)KerInteractionForcesFluidBox<. . .> (false,p1,pini,pfin,viscof,ftomassp,
tauff,posxy,posz,pospress,velrhop,temp,code,idp,CTE.massf,ftmassp1,ftp1,posdp1,posp1
,velp1,pressp1,rhopp1,tempp1,taup1_xx_xy,taup1_xz_yy,taup1_yz_zz,grap1_xx_xy,grap1_x
z_yy,grap1_yz_zz,acep1,arp1,atempp1,visc,deltap1,tshifting,shiftposp1,shiftdetectp1)
;
    . . .
    //-Stores results.
    if(shift||arp1||acep1.x||acep1.y||acep1.z||visc){
      . . .
      ar[p1]+=arp1;
      atemp[p1] += atempp1;
 . . .
```

This is a CUDA kernel, it is executed N times by N threads and is callable from CPU.

Update interaction forces functions



JSphGpu_ker.cu

```
__device__ void KerInteractionForcesFluidBox(. . .,const float4 *velrhop,const double
*temp,. . .,float rhopp1, double tempp1,. . .,float &arp1,float &atempp1,. . .){
  for(int p2=pini;p2<pfin;p2++){
    . . .
    if(rr2<=CTE.fourh2 && rr2>=ALMOSTZERO){
      //-Cubic Spline, Wendland or Gaussian kernel.
     float frx,fry,frz,fabc;
     if(tker==KERNEL_Wendland)KerGetKernelWendland(rr2,drx,dry,drz,frx,fry,frz,fabc);
      else
     if(tker==KERNEL_Gaussian)KerGetKernelGaussian(rr2,drx,dry,drz,frx,fry,frz,fabc);
      else if(tker==KERNEL_Cubic)KerGetKernelCubic(rr2,drx,dry,drz,frx,fry,frz,fabc);
      . . .
      //-Density derivative.
      const float dvx=velp1.x-velrhop2.x, dvy=velp1.y-velrhop2.y, dvz=velp1.z-
     velrhop2.z;
      if(compute)arp1+=(USE_FLOATING? ftmassp2: massp2)*(dvx*frx+dvy*fry+dvz*frz);
      . . .
     // Temperature: compute temperature derivative
     if (compute) {
         float heatKp2 = (boundp2 ? CTE.HeatKBound : CTE.HeatKFluid);
         float rhopp2 = (boundp2 ? CTE.DensityBound : velrhop2.w);
         const double dtemp = tempp1 - temp[p2]; // (dtemp=tempp1-tempp2)
         const float tempConst = (4 * massp2*CTE.HeatKFluid*heatKp2) /
(CTE.HeatCpFluid *rhopp1*rhopp2*(CTE.HeatKFluid + heatKp2));
         atempp1 += float(tempConst*dtemp*fabc);
     }
```

This is a device function, it is only callable from the GPU.

Update interaction forces functions

JSphGpu_ker.cu

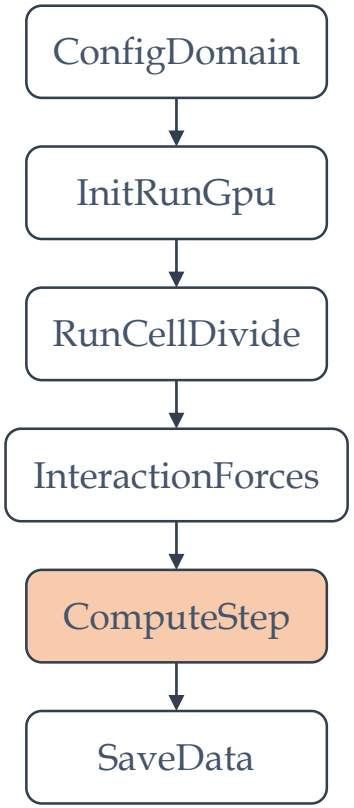
```
__global__ void KerInteractionForcesBound(. . .,const float4 *velrhop,const double
*temp,. . .,float *ar,float *atemp){
  unsigned p1=blockIdx.y*gridDim.x*blockDim.x + blockIdx.x*blockDim.x + threadIdx.x;
//-Number of particle.
  if(p1<n){
    float visc=0,arp1=0;
    float atempp1 = 0;
    //-Loads particle p1 data.
    . . .
    double tempp1 = temp[p1];
    . . .
    if(pfin)KerInteractionForcesBoundBox<psingle,tker,ftmode>
(p1,pini,pfin,ftomassp,posxy,posz,pospress,velrhop,temp,code,idp,CTE.massf,posdp1,posp
1,velp1,tempp1,arp1,atempp1,visc);
    . . .
    //-Stores results.
    if(arp1 || visc){
      ar[p1]+=arp1;
      atemp[p1] += atempp1;
    . . .
```

Update interaction forces functions

JSphGpu_ker.cu

```
__device__ void KerInteractionForcesBoundBox (. . .,const float4 *velrhop,const double
*temp,. . .,float3 velp1,double tempp1,float &arp1,float &atempp1,float &visc){
  for(int p2=pini;p2<pfin;p2++){
    . . .
    if(rr2<=CTE.fourh2 && rr2>=ALMOSTZERO){
      float frx,fry,frz,fabc;
      if(tker==KERNEL_Wendland)KerGetKernelWendland(rr2,drx,dry,drz,frx,fry,frz,fabc);
      else if(tker==KERNEL_Gaussian)
KerGetKernelGaussian(rr2,drx,dry,drz,frx,fry,frz,fabc);
      else if(tker==KERNEL_Cubic) KerGetKernelCubic(rr2,drx,dry,drz,frx,fry,frz,fabc);
      . . .
    if(compute){
      //-Density derivative.
      const float dvx=velp1.x-velrhop2.x, dvy=velp1.y-velrhop2.y, dvz=velp1.z-
velrhop2.z;
      arp1+=(USE_FLOATING? ftmassp2: massf)*(dvx*frx+dvy*fry+dvz*frz);
      //==============================================
      // Temperature: compute temperature derivative
      const double dtemp = tempp1 - temp[p2]; // Temperature: (dtemp=tempp1-tempp2)
      const float tempConst = (4 * ftmassp2*CTE.HeatKFluid*CTE.HeatKBound) /
(CTE.HeatCpBound*CTE.DensityBound*velrhop[p2].w*(CTE.HeatKFluid + CTE.HeatKBound));
      atempp1 += float(tempConst*dtemp*fabc);
      //==============================================
. . .
```

Compute Verlet



JSphGpu_ker.cu

```
template<bool floating,bool shift> __global__ void KerComputeStepVerlet(unsigned
n,unsigned npb,float rhopoutmin,float rhopoutmax,const float4 *velrhop1, const
float4 *velrhop2,const double *temp1, const double *temp2,const float *ar,const
float *atemp,const float3 *ace,const float3 *shiftpos,double dt,double
dt205,double dt2
,double2 *movxy,double *movz,typecode *code,float4 *velrhopnew,double *tempnew){

  unsigned p=blockIdx.y*gridDim.x*blockDim.x + blockIdx.x*blockDim.x +
threadIdx.x; //-Number of particle.
  if(p<n){
    if(p<npb){ //-Particles: Fixed & Moving.
      float rrhop=float(double(velrhop2[p].w)+dt2*ar[p]);
      rrhop=(rrhop<CTE.rhopzero? CTE.rhopzero: rrhop); //-To prevent absorption of
fluid particles by boundaries. | Evita q las boundary absorvan a las fluidas.
      velrhopnew[p]=make_float4(0,0,0,rrhop);
      tempnew[p] = temp2[p]; // Temperature: constant temperature on boundaries
for this implementation.
    }else{ //-Particles: Floating & Fluid.
      tempnew[p] = temp2[p] + dt2 * atemp[p]; // Temperature: update temperature
      . . .
```

Compute Symplectic

JSphGpu_ker.cu

```
template<bool floating,bool shift> __global__ void KerComputeStepSymplecticPre
(unsigned n,unsigned npb,const float4 *velrhoppre,const double *temppre,const float
*ar,const float *atemp,const float3 *ace,const float3 *shiftpos,double dtm,float
rhopoutmin,float rhopoutmax,typecode *code,double2 *movxy,double *movz,float4
*velrhop,double *temp){
  unsigned p=blockIdx.y*gridDim.x*blockDim.x + blockIdx.x*blockDim.x + threadIdx.x; //-
Number of particle.
  if(p<n){
    if(p<npb){ //-Particles: Fixed & Moving.
      . . .
      velrhop[p]=rvelrhop;
      temp[p] = temppre[p]; // Temperature: does not change in the boundaries.
    }else{ //-Particles: Floating & Fluid.
     //-Updates density.
     float4 rvelrhop=velrhoppre[p];
     rvelrhop.w=float(double(rvelrhop.w)+dtm*ar[p]);
     temp[p] = temppre[p] + dtm * atemp[p]; // Temperature: Calculate new temperature
for the fluid
     . . .
```

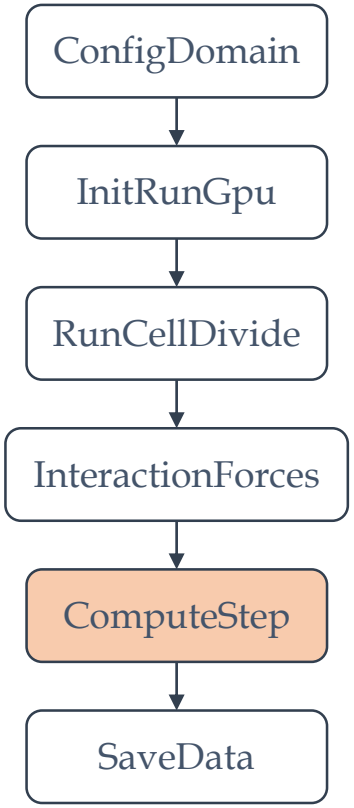
Compute Symplectic

JSphGpu_ker.cu

```
template<bool floating,bool shift> __global__ void KerComputeStepSymplecticCor
(unsigned n,unsigned npb,const float4 *velrhoppre,const double *temppre,const float
*ar,const float *atemp,const float3 *ace,const float3 *shiftpos,double dtm,double
dt,float rhopoutmin,float rhopoutmax,typecode *code,double2 *movxy,double *movz,float4
*velrhop,double *temp){

  unsigned p=blockIdx.y*gridDim.x*blockDim.x + blockIdx.x*blockDim.x + threadIdx.x; //-
Number of particle.
  if(p<n){
    if(p<npb){ //-Particles: Fixed & Moving.
      . . .
    }else{ //-Particles: Floating & Fluid.
      //-Updates density.
      double epsilon_rdot=(-double(ar[p])/double(velrhop[p].w))*dt;
      float4 rvelrhop=velrhoppre[p];
      rvelrhop.w=float(double(rvelrhop.w) * (2.-epsilon_rdot)/(2.+epsilon_rdot));
      //================================================
      // Temperature
      //================================================
      const double epsilon_tdot = (-double(atemp[p]) / temp[p])*dt;
      temp[p] = temppre[p] * (2. - epsilon_tdot) / (2. + epsilon_tdot);
      //================================================
      . . .
```

Free memory

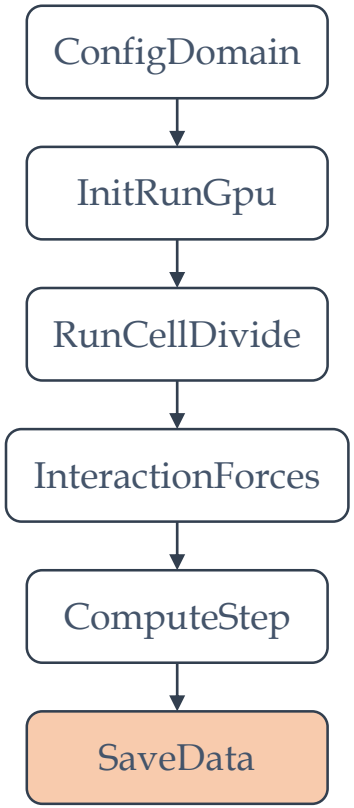


JSphGpu_ker.cu

```
void JSphGpu::PosInteraction_Forces(){
    //-Frees memory allocated in PreInteraction_Forces().
    ArraysGpu->Free(Arg); Arg=NULL;
    ArraysGpu->Free(Aceg); Aceg=NULL;
    ArraysGpu->Free(ViscDtg); ViscDtg=NULL;
    ArraysGpu->Free(Deltag); Deltag=NULL;
    ArraysGpu->Free(ShiftPosg); ShiftPosg=NULL;
    ArraysGpu->Free(ShiftDetectg); ShiftDetectg=NULL;
    ArraysGpu->Free(PsPospressg); PsPospressg=NULL;
    ArraysGpu->Free(SpsGradvelg); SpsGradvelg=NULL;

    ArraysGpu->Free(Atempg);     Atempg = NULL;
}
```

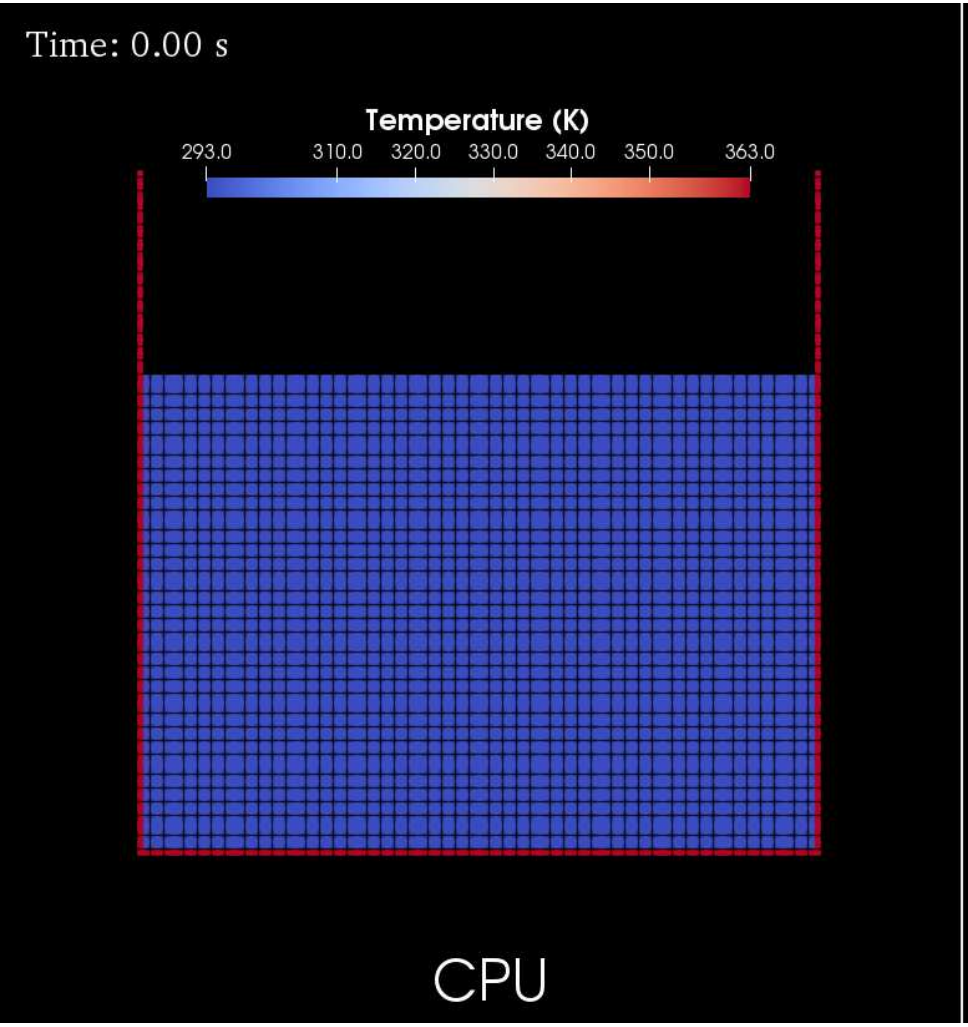
Copy data from GPU to save data

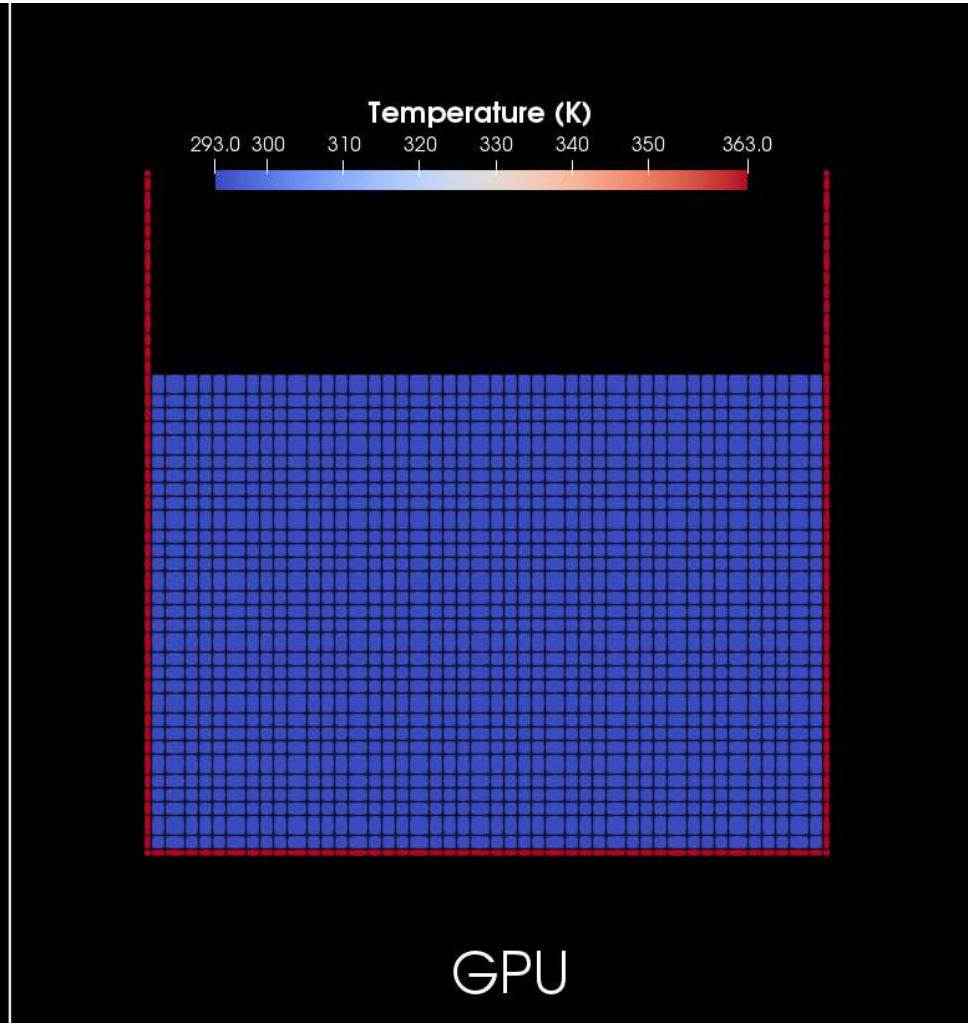


JSphGpu.cpp

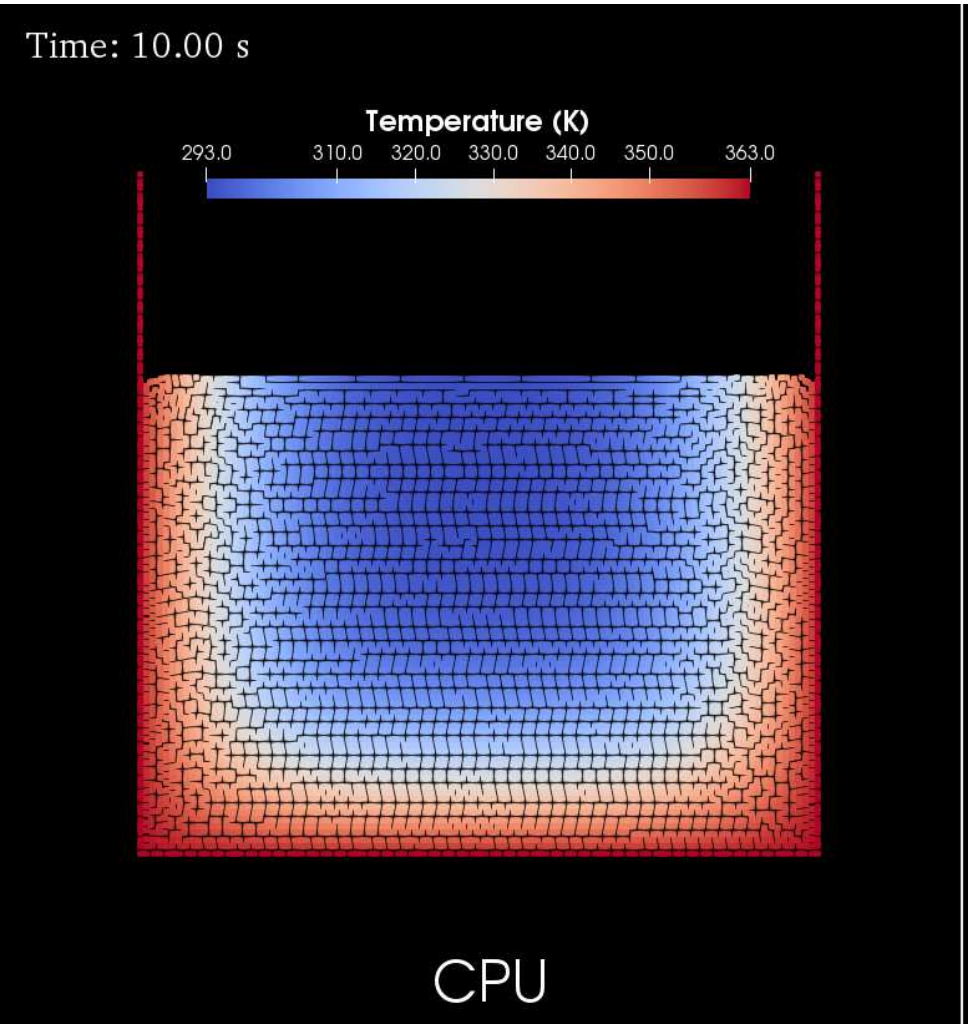
```
unsigned JSphGpu::ParticlesDataDown(unsigned n,unsigned pini,bool code,bool
cellorderdecode,bool onlynormal){
. . .
cudaMemcpy(Velrhop,Velrhopg+pini,sizeof(float4)*n,cudaMemcpyDeviceToHost);
cudaMemcpy(Temp, Tempg + pini, sizeof(double)*n, cudaMemcpyDeviceToHost);
. . .
//-Eliminates abnormal particles (periodic and others).
if(onlynormal){
  unsigned ndel=0;
  for(unsigned p=0;p<n;p++){
    const bool normal=CODE_IsNormal(Code[p]);
    if(ndel && normal){
      . . .
      Velrhop[p-ndel]=Velrhop[p];
      Temp[p-ndel] =Temp[p];
      . . .
//-Converts data to a simple format.
for(unsigned p=0;p<n;p++){
  . . .
  AuxRhop[p]=Velrhop[p].w;
  AuxTemp[p]=Temp[p];
}
. . .
```

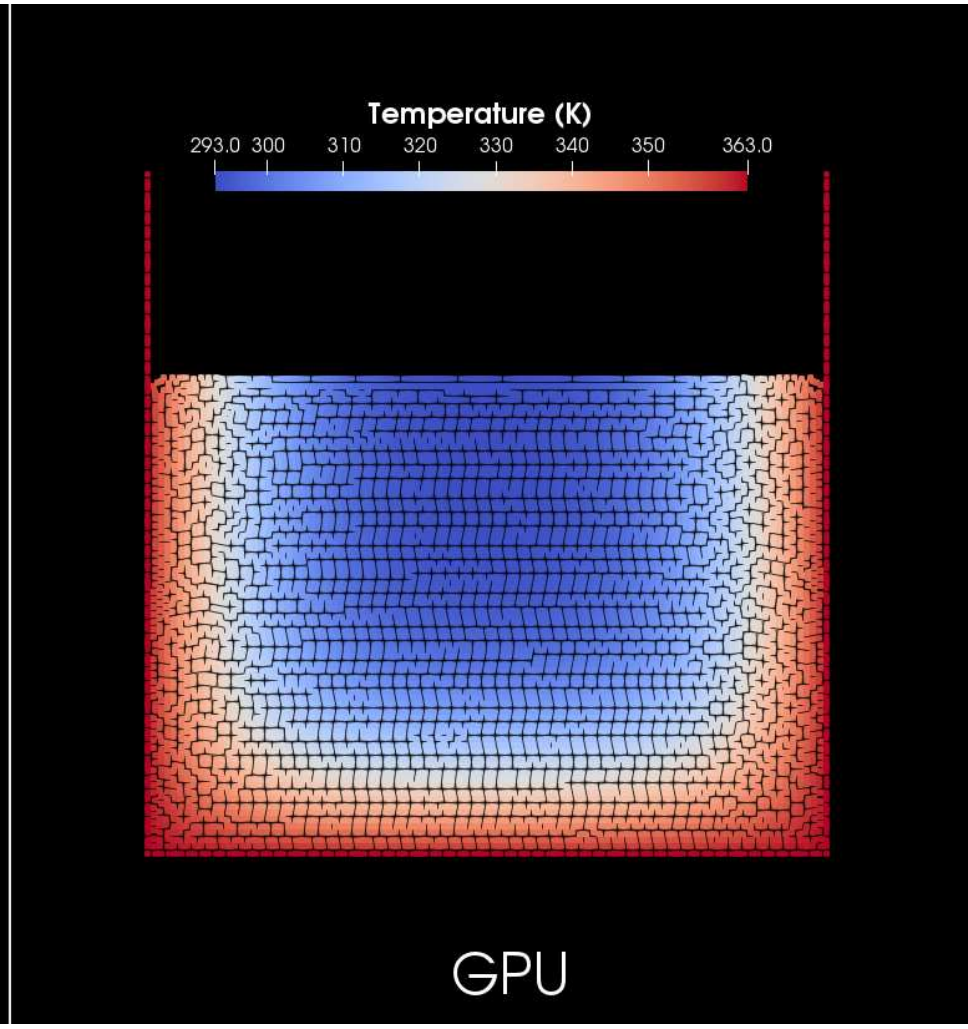
Test and compare GPU implementation



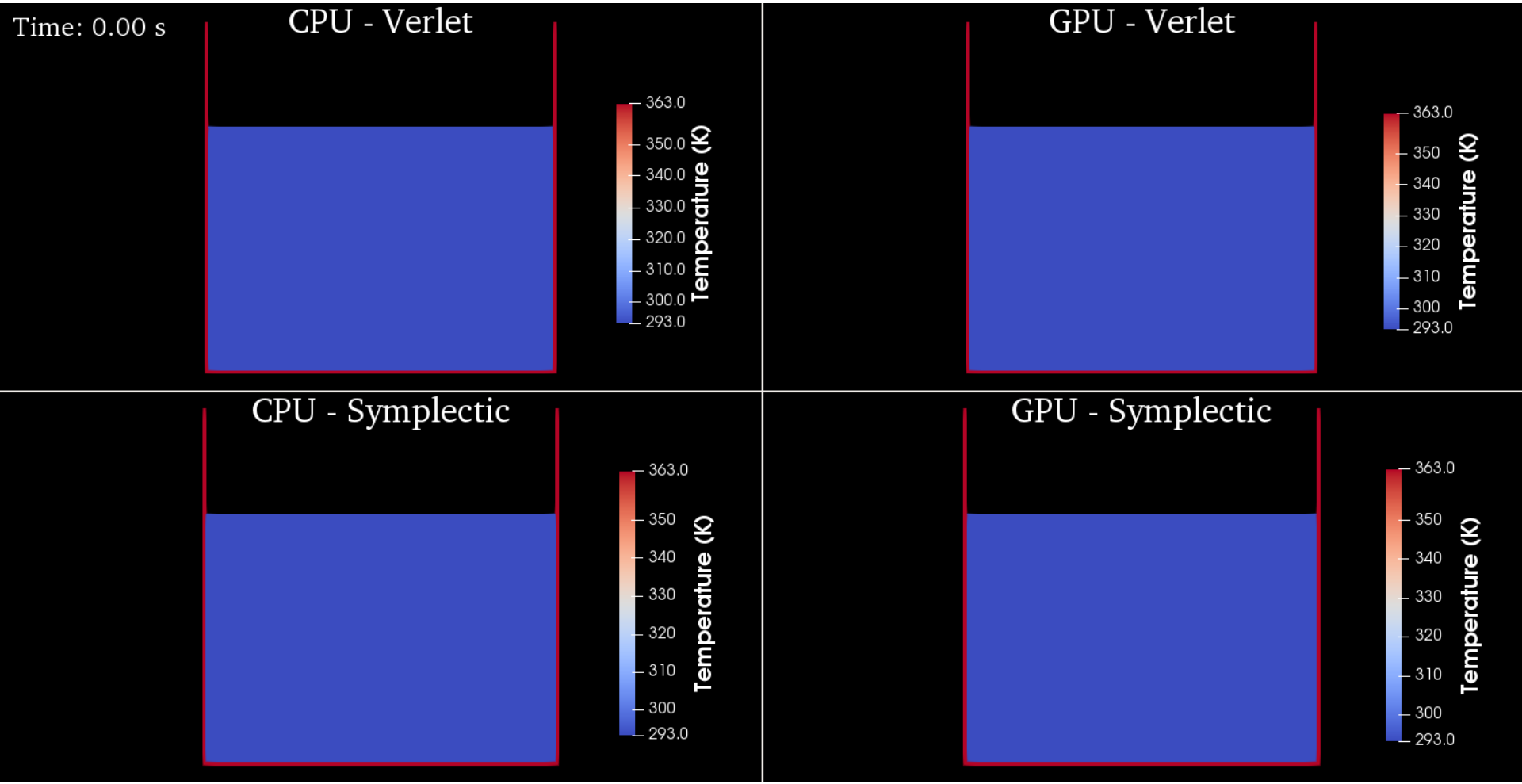


Test and compare GPU implementation

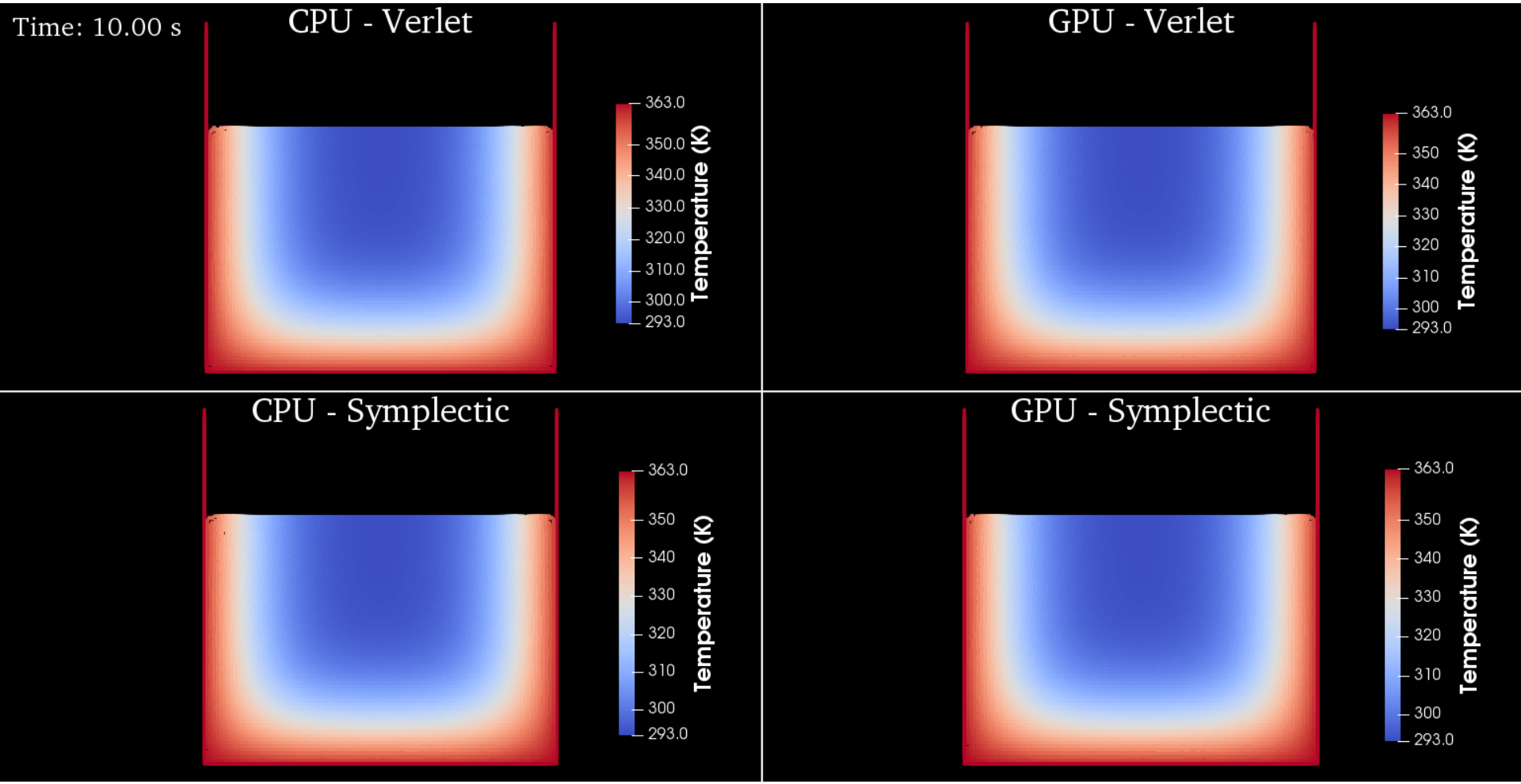




Compare Verlet and Symplectic schemes



Compare Verlet and Symplectic schemes



Conclusions

We have not modified pre-processing tools to add new configuration parameters.

We have not modified post-processing tools to add new variables.

There are tons of files but it's only necessary to modify a few of them.

It's easier to develop on CPU first and then on GPU.

To add a new feature, just look for a similar one in the code and follow the procedure.

Thanks for your attention ☺

Developing on DualSPHysics:

examples on code modification and extension

O. García-Feal [orlando@uvigo.es], L. Hosain, J.M. Domínguez, A.J.C. Crespo

4th DualSPHysics Users Workshop · Instituto Superior Técnico, University of Lisbon 22nd-24th October 2018