Accelerating multi-dimensional population balance model simulations via a highly scalable GPU framework.

Chaitanya Sampata , Yukteshwar Baranwala , Rohit Ramachandrana,*∗*

*a Chemical and Biochemical Engineering, Rutgers University, Piscataway, NJ, USA - 08854*

**Abstract**

Population balance models (PBMs) are widely used to sim- ulate and optimize processes that are distributed with respect to their internal and spatial coordinates/properties. Prediction of the time evolution of the distribution of these properties using these PBMs, is very important in understanding key dynamics. However, with an increase in the number of components and phases in a particulate mixture, the complexity of PBM increases. This leads to multi-dimensional matrix calculations potentially requiring significant computational power. Solving such a system of equations with a traditional central processing unit (CPU) framework would be intractable. This study focuses on the development of a novel algorithm to parallelize the nested loops inside the PBM via a GPU framework. The communication time is much lower in comparison to the speedup achieved in the parallel section of the code on the GPUs. The speed up achieved was significant compared to the PBM code when run in serial or in on multi-core configuration. The speed improvements for the code for various CPU & GPU architectures and configurations have been reported. (give some metrics).

*∗*Corresponding author

*Email address:* [rohitrr@soemail.rutgers.com](mailto:rohitrr@soemail.rutgers.com) (Rohit Ramachandran)

*Preprint submitted to Computer and Chemical Engineering November 7, 2019*

*Keywords:* Population Balance Model, GPU, Parallel Computing,

Granulation, MPI

1 **1. Introduction & Objectives**

I THINK THIS SECTION SHOULD BE NON-PHARMA FOCUSSED. IT SHOULD TALK ABOUT PBMS AND TH EIR COMPUTATIONAL EXPENSE ETC. TOWARD THE END YOU CAN MENTION GRANULATION AS AN APPLICATION.

2

12 START FROM HERE AND EXPAND. Over the past decade, Population

20 Balance Models (PBMs) have been used to predict the behavior of granulation

21 processes (Barrasso et al., 2013),(Ramachandran et al., 2009).

22 PBMs are used to calculate bulk rate processes occurring during granulation.

23 On various occasions they are unable to capture some physics-based process in-

24 formation, thus a mechanistic kernel can be introduced in these models to make

25 them more accurate. Another way to increase its accuracy, is to incorporate

26 larger number of solid bins inside the PBM. The increase in the number of solid

27 bins leads to an increase in calculations for each time step, leading to a higher

28 simulation times. The calculations increase by a factor of 2*n* where n being

29 the number of solid bins. An accurate model which incorporates higher num-

30 ber of solid bins as well as includes a mechanistic kernel in its calculations is

31 expected to be sluggish to simulate and could take several hours to complete.

32 Such models and their solving techniques are not viable to be used in real time

33 system control. Thus, there is a need to improve the time it takes to simulate

34 the model.

35 The advancement of computers and its peripherals in recent years have led

36 to a great increase in computational resources leading to faster simulations. The

37 recent central processing unit (CPU) now contain various cores thus making it

38 possible to run multiple processes in parallel. In order to take advantage of a

39 highly parallel framework, large number of cores are needed which may not be

40 possible in a personal desktop and a supercomputer cluster needs to be used.

41 Another computer peripheral that can to be used to run a highly parallel code

42 is the computer’s graphic processing unit (GPU) (Prakash et al., 2013). These

43 GPUs contain thousands of compute cores that can be used run tasks in parallel.

44 Thus, a desktop equipped with a GPU could compute the same results as a CPU

45 code on supercomputers in lesser amount of time as seen in Section 4. With

46 the launch of Compute Unified Device Architecture (CUDA), NVIDIA made

47 it easier to use GPUs for general parallel programming in an approach usually

48 termed as general purpose computing on GPUs (GPGPUs).

1.1 Objectives (MAKE IT MORE LIKE OBJECTIVES. NO NEED TO MENTION HERE HOW YOU ARE GOING ABOUT IT. You seem to describe more the emethod. https://www.sciencedirect.com/science/article/pii/S0098135418303594

49

The main objective of this study is to develop a GPU-parallelized highly scalable multi-scale PBM.

n the present study, a mechanistic multi-dimensional PBM was developed

50 such that it was not only accurate but also scalable since the number of solid

51 bins could be changed to alter its behaviour. This model was developed in C++

52 to be run on CPUs. This C++ model was paralleliZed using Message Parsing

53 Interace (MPI) which is parallel application programming interface (API). This

54 model was developed NVIDIA GPUs and was parallelized using the CUDA

55 toolkit. The timings of the simulations were then compared for each of these

56 cases. The scalability of the GPU based code was also tested to obtain speed

57 improvements over serial CPU code.

58 **2. Background**

59 *2.1. Granulation and population balance modeling*

60 start with PBms… granulation at the end. Granulation is the process of engineering granules from pharmaceutical pow-

61 der blends with the addition of liquid or solid binders. This process is usually

62 carried out to obtain granules with a certain PSD, bulk densities and other

63 physical properties (Barrasso and Ramachandran, 2015). There are about 3

64 rate processes that occur due the addition of a liquid binder to the powder

65 mixture are wetting and nucleation, consolidation and aggregation, and break-

66 age and attrition (Sen et al., 2014). In a high shear granulator, the particles

67 are rendered wet when they come in contact with the liquid binder, which also

68 aids in granule formation due to liquid bridges. These granules can also break

69 into smaller fragments due to shear stresses, compressive and tensile forces that

70 are exerted on to the system due the impeller, particle-particle interactions and

71 particle-wall interactions.

72 To understand the process dynamics and control these process, population

73 balance equations have been accepted as a relevant methodology (Immanuel and

74 Doyle, 2005). Population balances have been successful in predicting physical

75 phenomena occurring in granulation such as aggregation, breakage and consol-

76 idation. These models predict how groups of distinct entities inside the phar-

77 maceutical powder behave on a bulk scale due to process parameters over time

78 of granulation. A general representation of the model is:

*∂*

*F* (**v***,* **x***, t*) +

*∂t*

*∂ d***v**

[*F* (**v***,* **x***, t*)

*∂***v** *dt*

(**v***,* **x***, t*)] +

*∂ d***x**

[*F* (**v***,* **x***, t*)

*∂***x** *dt*

(**v***,* **x***, t*)]

= *SRf ormation* (**v***,* **x***, t*) + *SRdepletion* (**v***,* **x***, t*) + *F*˙*in* (**v***,* **x***, t*) *− F*˙*out* (**v***,* **x***, t*) (1)

79 where **v** is a vector of internal coordinates. **v** is commonly used to describe

80 the solid, liquid, and gas content of each type of particle. The vector **x** repre-

81 sents external coordinates, usually spatial variance. *F* represents the number of

82 particles present inside the system, *F*˙*in* and *F*˙*out* is the rate of particles coming

83 in and going out the system respectively. *SRf ormation* and *SRdepletion* are the rate

84 of formation and depletion due various phenomena occurring in granulation.

85 Prediction of PSDs and other particle bulk properties is highly dependent on

86 the kernels used to describe the sub-processes inside granulation. Identification

87 of a kernel that describes the sub-processes suitably is of the essence in this

88 model since they are not only size dependent but also time dependent. Recently,

89 various mechanistic kernels have been developed that help capture the micro-

90 mechanics of the system, which help in better prediction of the final particle

91 size (Barrasso et al., 2015b). Several of these kernels maybe dependent on other

92 physical simulations such as Discrete Element Model (DEM) to obtain certain

93 information about the microscopic phenomena occurring inside the system. I think its important to talk about the kernels. The kernels (esp aggregation ) is responsible for most of the simulation time. What does DEM have to do with the time…lets discuss.

94 *2.2. Parallel Computing*

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

Computing with its current infrastructure has become an important tool in science. Parallel computing is one of the more extensively used type of comput- ing used by scientists to perform simulations. It is the process of splitting of larger calculations into many smaller processes executed concurrently (Almasi and Gottlieb, 1989). This type of execution helps achieve large speed gains overs simulations run on a single core in a serial manner. The computational task can be decomposed by various means to help simulate the system in a reasonable amount of time. The simulation problem can be decomposed either at the task level or at the data level. Task parallelism involves each process to behave distinctively from another as they would each be performing different operations. These multiple operations could be performed on a single data set or on multiple data sets, known as multiple instruction single data (MISD) and multiple instruction multiple data (MIMD) respectively. On the other hand, data parallelism involves the distribution of data across various processes which usually perform same set of operations on the data (Solihin, 2015). This type of parallelism is known as single instruction multiple data (SIMD). MIMD and SIMD can also be combined in certain systems, thus decreasing the simulation times further.

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

*2.3. GPU based parallel computing*

Traditionally, large parallel jobs needed to be run on supercomputers which had thousands of cores, but these are require special components making them expensive. Graphic processing unit (GPU) were initially used for vector calcu- lations to support graphics inside a computer system. But, lately GPU manu- facturers have started to promote them general computing as well. This form of computing has been gaining popularity among scientists to accelerate sim- ulations (Kandrot and Sanders, 2011). These GPUs comprise of a massively parallel architecture with hundreds to thousands of computational cores which can have thousands of active threads running simultaneously (Keckler et al.,

2011). This means that GPUs have large cWomputing potential the GPU com- puting can be exploited using parallel programming languages such as OpenCL and CUDA.

CUDA is an application programming interface (API) developed by NVIDIA (NVIDIA Corporation, 2012) that enables users to program parallel code for execution on the GPU. This framework is an extension implemented on top of C/C++ or Fortran. Parallel code for the GPU is written as kernels, which theoretically are similar to functions or methods in traditional programming languages. As several parts of the code need to executed only once during a simulation, only few sections of the code can be written in terms of kernel while the remaining has to be executed in serial on the CPU of the system. The nvcc compiler from the CUDA toolkit prioritizes the compilation of these kernels before passing the serial section of the code to the native C/C++ compiler inside the system. There are three main parallel abstractions that exist in CUDA are grids, blocks and threads (Santos et al., 2013). Each CUDA kernel is executed in a serial manner during the execution of the program unless specified, where the kernels can be run in parallel using CUDA streams. Each kernel executes as a grid which in turn consists of various blocks which are consistuted by various threads. This thread-block-grid hierarchy helps obtain fine grained data level and thread level parallelism. An illustration of this hierarchy is observed in Figure 1.

144

145

146

147

148

149

150

151

152

153

154

155

Another important aspect related to GPU parallelization is the data com- munication between the threads. The GPU consists of various memory modules with different access limitations as shown in Figure 1. The threads inside each block can communicate with each other using the shared memory. This memory is local to the block where these threads exist i.e. they are not accessible by threads from other blocks. In addition to the shared memory each thread has its own local memory where local/temporary variables for each kernel can be saved to them. The threads from different communicate with each other using the global memory which is visible to all blocks inside the GPU at the cost of higher communication times. Accessing of data from the local memory is the fastest for a thread and its slows down as we move towards shared block memory

and the least for accessing data from the global GPU memory.

156

*2.4. Previous parallelized PBM works*

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

PBMs have known to be been computationally intensive especially ones with larger internal coordinates and higher dimensions. Thus, several researchers have made attempts to increase the speed of these simulations. Gunawan et al. (2008) developed a parallelization technique for a high-resolution finite volume solution of the PBM. The parallel algorithm presented by Gunawan et al. (2008) in their scale up studies for the PBM upto 100 cores achieved good parallel efficiency, implying the algorithm’s effectiveness. The study was not performed for higher number of dimensions inside the PBM. This study also suggested that an algorithm with a shared memory model could help improve simulation speeds further. A hybrid memory model was implemented by Bettencourt et al. (2017) to obtain speed improvements of about 98% from the serial code. This implementation took into account both Message Parsing Interface (MPI) as well as Open Multi-processing (OMP). A similar PBM parallelization approach was also undertaken in (Sampat et al., 2018) and a speed up of about 13 was obtained. The reduction in speed was attributed to the use of dynamic arrays used in their PBM framework to accommodate the hybrid nature of the model being used.

Multiprocessor (Grid)

Block(2,n)

Block

(0,0)

Block

(0,n)

Shared memory

Block

(1,0)

Block

(2,0)

Block

(1,n)

Block

(2,n)

Thread (0,0)

Thread

(0,n)

Thread

(m,0)

Thread

(m,n)

Block

(m,0)

Block

(m,n)

Local

Memory

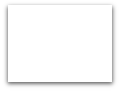
Thread (m,n)

Global Memory

Constant Memory

Texture Memory

Figure 1: The parallel structure matrix inside the GPU and the various memories associated



with each structure.

174

175

176

177

178

179

180

181

182

Algorithms to parallelize the PBM codes on GPU have been studied briefly by Prakash et al. (2013) using the inbuilt MATLAB’s parallel computing toolbox (PCT). This study was able to achieve good speed ups but could have been higher if the code had been implemented in native programming languages such as C or FORTRAN. Other works that have used GPU acceleration to improve computation times for their population balance simulations include those from various other chemical engineering processes such as crystallization (Szil´agyi and Nagy, 2016) , combustion (Shi et al., 2012) , multiphase flow (Santos et al.,

2013) , coagulation dynamics (Xu et al., 2015)

183

**3. Method and implementation**

184

*3.1. PBM implementation*

Population balance model in its general formed is expressed as follows (Ramkr- ishna and Singh, 2014):

*d*

*dt F* (*si , l, g, x*) = *SRagg* (*si , l, g, x*) + *SRbreak* (*si , l, g, x*)

+ *F*˙*in* (*si , l, g, x*) *− F*˙*out* (*si , l, g, x*) (2)

185

186

187

188

189

190

191

where, *F* (*si , x*) represents the number of solid particles of type i being stud- ied in each spatial compartment *x* of the granulator. The rate of aggregation *SRagg* (*si , x*) and the rate of breakage *SRbreak* (*si , x*) determines the rate at which particles density changes within different size classes. The rate of particles enter-

ing, *F*˙*in* (*si , x*) and exiting, *F*˙*out* (*si , x*), the spatial compartment due to particle

transfer also affects their number in each size class. The rate of change of internal liquid volume in each particle can be calculated as:

*dt F* (*si , x*)*l*(*si , x*) = *SRliq,agg* (*si , x*) + *SRliq,break* (*si , x*) + *F*˙*in* (*si , x*)*lin* (*si , x*)

*d*

*− F*˙*out* (*si , x*)*lout* (*si , x*) + *F* (*si , x*)*l*˙*add* (*si , x*) (3)

192

193

194

195

196

197

198

199

where, *l*(*si , x*) is the internal liquid volume in each particle with *si* as the solid volume for solid type 1 in the spatial compartment *x*. *SRliq,agg* (*si , x*) and *SRliq,break* (*si , x*) are the rates at which liquid is transferred between size classes due to aggregation and breakage respectively. *lin* (*si , x*) and *lout* (*si , x*) are the internal liquid volumes of the particles entering and exiting the spatial com- partment. *ladd* (*si , x*) is the volume of liquid acquired by each particle in the compartment at every time step due to external liquid addition. Similarly, the

rate of change of gas volume is calculated using the following equation:

*dt F* (*si , x*)*g*(*si , x*) = *SRgas,agg* (*si , x*) + *SRgas,break* (*si , x*) + *F*˙*in* (*si , x*)*gin* (*si , x*)

*d*

*− F*˙*out* (*si , x*)*gout* (*si , x*) + *F* (*si , x*)*gc o*˙*ns*(*si , x*) (4)

200

201

202

203

204

205

206

207

where, *g*(*si , x*) is the gas volume of each particle with solid volumes of *si* in the spatial compartment *x*. *SRgas,agg* (*si , x*) and *SRgas,break* (*si , x*) are the rates of gas transferred between size classes due to aggregation and breakage respectively. *gin* (*si , x*) and *gout* (*si , x*) are the gas volume of the particles entering and leaving the spatial compartment respectively. *gc o*˙*ns*(*si , x*) represents the volume of gas particles formed due to process of consolidation occurring inside the system. The rate of aggregation, *SRagg* (*si , x*) in Equation 2 is calculated as (Chaturbedi

et al., 2017):

1 *s*1 *s*2

*SRagg* (*si , x*) = 2

*β*(*s , si − s , x*)*F* (*si , x*)*F* (*si − s , x*)*ds ds*

*i i i* 1 2

0 0

*− F* (*si , x*)

*smax*1 *−s*1 *smax*2 *−s*2

*β*(*si , s , x*)*F* (*s , x*)*ds ds*

(5)

*i i* 1 2

0 0

*i*

208

209

where, *β*(*si , s , x*) is the aggregation kernel and is expressed as a function of collision frequency (*C* ) and collision efficiency (*ψ*) (Barrasso et al., 2015a):

*β*(*si , s , x*) = *C* (*si , s , x*)*ψ*(*si , s , x*) (6)

*i i i*

210

211

212

213

214

215

216

The collision frequency of the solid particles was evaluated from the existing DEM data from (Sampat et al., 2018). To facilitate this study, it was assumed that the collision frequency was independent of the liquid particles present in the system.

The collision efficiency *ψ* was estimated based on Stokes , which states that a collision is successful when the Stokes number *Stv* associated with the collision

is lesser than the critical Stokes number *St∗* for the particles. These number

*v*

217

are calculated as follows:

*Stv* =

8*m*˜ *U*

3*πd*˜2 *µ*

(7)

*St∗* =

*v*

1

1 + *log e*

*h*

*ha*

(8)

218

Here, *m*˜

& *d*˜ represent the harmonic mean of the masses and diameters of

219

220

221

222

223

224

the particles respectively.*U* is the collision velocity, *µ* is the viscosity of the system and *e* is the coefficient of restitution. The thickness of the liquid on the surface of the particle *h* and the height of surface asperities *ha* were obtained from (Barrasso et al., 2015a). *Uc ritical* is defined as the ratio of the critical Stokes number to the Stokes number associated with the collision. The collision frequency *ψ* is defined as:

*Uc ritical*

Ψ =

0

*p*(*U* )*dU* (9)

225

where it is assumed that the collision velocities follow a log normal distribution:

*p*(*U* ) =

1

*U √*2*πσ*

exp

r (*lnU − µ*)2 l

*−* 2*σ*2

(10)

Similarly, the breakage rate can be expressed as follows:

*SRbreak* (*si , x*) =

*smax*1

0

*smax*2

0

*Kbreak* (*s*1 *, s*2 *, x*)*F* (*s*1 *, s*2 *, x*)*ds*1 *ds*2

*− Kbreak* (*si , x*)*F* (*si , x*) (11)

226

where, the breakage kernel *Kbreak* (*si , x*) is formulated as:

*Kbreak* (*si , x*) = *Cimpact*

*∞*

*Ubreak*

*p*(*U* )*dU* (12)

227

228

229

Similar to the aggregation kernel, *Ci mpact* is defined as rate at which the par- ticles impact with the geometry in the DEM simulation. Critical velocity for breakage to occur is defined as:

*Ub reak* =

*d*

2*St∗ef* 9 (1 *−* )2 9*µ*

(13)

*ρsi* 8

*d*

16*dpi*

230

where 2*St∗ef* is defined as critical Stokes deformation number (Iveson et al.,

232

The rate of increase of liquid volume of a particle, *l*˙*add* (*si , x*) is expressed as:

*l*˙*add* (*si , x*) =

(l:*i* (*si* )(*m*˙ *spray* )

*msolid* (*x*)

(14)

233

234

235

236

237

where, l:*i* (*si* ) is the total solid volume of the particle; *m*˙ *s pray* is the rate of external liquid addition and *msolid* is the total amount of solid present in the compartment.

The rate of decrease in gas volume per particle due to consolidation is cal- culated using the following expression (Verkoeijen et al., 2002):

*g*˙*cons* (*si , x*) =*c*(*νimpeller* )*ω V* (*si , x*)

r

(1 *− min* )

*s*

*min* l

*g*(*si , x*) + *l*(*si , x*) *−* (*s*1 + *s*2 ) 1 *−*

*min*

(15)

238

239

240

where, *c* and *ω* are the consolidation constants; *vimpeller* is the impeller rotational speed; *min* is the minimum porosity;

Particle transfer rate, *F*˙*out* (*si , x*) in Equation 2 is calculated as:

*F*˙*out* (*si , x*) = *F*˙ (*si , x*)

*νcompartment* (*x*) *∗ dt dcompartment*

(16)

241

242

243

244

245

246

247

248

249

250

where, *νcompartment* (*x*) and *dcompartment* are respectively the average velocity of particles in compartment *x* and the distance between the mid-points of two adjacent compartment, which is the distance particles have to travel to move to the next spatial compartment. *dt* is the time-step.

A finite difference method was used to solve the developed system of ordi- nary differential equations (ODEs) (Barrasso and Ramachandran, 2015). The numerical integration technique used was first order Euler integration as it is commonly used for speed improvements as while having minimal impact on ac- curacy (Barrasso et al., 2013). In order to avoid numerical instability due to the explicit nature of the Euler integration, Courant-Friedrichs-Lewis (CFL) con-

252

253

254

each iteration such that, the number of particles leaving a particular bin at any time is less than the number of particles present at that time (Ramachandran

and Barton, 2010).

255

*3.2. MPI implementation*

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

274

The message parsing interface (MPI) parallel implementation of the PBM was focused towards equal distribution of the task and memory. The imple- mentation used in this work differs from the hybrid implementation used by (Bettencourt et al., 2017) and (Sampat et al., 2018) as only MPI was used to parallelize the code. It was pointed by (Sampat et al., 2018) that open mes- sage parsing (OMP) does not provide significant speed improvements due to limitations with usage of dynamic vectors which are essential for such a sys- tem. Thus, the OMP implementation was avoided which also meant that lesser number of cores would be required. The focus of this study to localize the com- putation power rather than depend on supercomputers/clusters. MPI was used for message passing from one core to another as well as all the calculations were performed on the same MPI core. As pseudo code has been presented in Algo- rithm 1 to illustrate the distribution of tasks. For each time step, a MPI process is responsible for a certain section of the problem, usually a spatial chunk inside the geometry (also referred to as compartment).

Simulations for this study were performed on a computer with an Intel Core i7-7700K processor clocked at 4.2GHz and 32 GB of RAM. For maximum per- formance while data reading and writing a SSD was used. GCC version 7.4 with

openMPI 2.0 was used to compile the parallel C++ code.

275

*3.3. GPU implementation*

276

277

278

279

280

NVIDIA’s CUDA toolkit extends the C language such that user defined functions called kernels can be created to be run on the GPU. These kernels can be executed several number of times in parallel using large number of threads. A thread is sequence if programmed instructions that can be managed by the computer’s scheduler. A kernel depending upon the dimensions of the data

**Algorithm 1** CPU-based Parallel Population Balance Model

1: **procedure** PBM(*NC omp* ,*NM P I* ) *1> NC omp* is the number of compartments

2: Divide *NC omp* in *NM P I*

3: **while** *t < tf inal* **do**

4: **for** *∀nC omp* in 1 MPI process **do**

5: Calculate *'!Raggregation* for solid bins *s*1 ,*s*2

6: Calculate *'!Rbreakage* for solid bins *s*1 ,*s*2

7: Calculate *nparticles* using Euler’s method

8: **end for**

9: Collect *nparticles* from *NM P I 1>* Master process collects all data

10: Calculate *timestep* using *CFL* condition

11: *tnew* = *t* + *timestep*

12: **end while**

13: **end procedure**

281

282

283

284

285

286

287

288

289

290

291

292

293

294

295

can execute instructions in 1-D, 2-D or 3-D thread blocks. The kernel can also launch multiple thread blocks at once, thus increasing the number of parallel process executions known as grid. Similar to a thread block, a grid can be up to 3-D depending upon the data under study. The code execution was split between the CPU (also called host) and GPU (also called device). Time sensitive calculations as well as mixed data calculations were handled on the CPU with a single core, while the more computationally intensive tasks were distributed on to the GPU using kernels. Like in the CPU parallelization, the geometry was split into multiple compartments. These compartments in turn formed the number of blocks inside each GPU kernel. The number of solids used helped formed the threads in each of these blocks. The work flow of the execution can be found in Figure 2. The orange arrows in Figure 2 indicate the transfer of data from the CPU memory to the GPU memory and vice-versa whereas the blue arrows indicate the sections of the code that is sent to the GPU for parallel

execution.

296

**4. Results and discussions**

297

298

299

Parallel efficiency of an algorithm can be tested either by strong scaling where the problem size remains the same and number of processing elements are increased or by weak scaling where the number of processing elements remain the

CPU (Host) GPU (Device)

Start

PBM grid variables

Allocate memory and copy

PBM grid variables - GPU copy

Initialize grid matrices

Kernel launch command

Copy calculated values

Calculate Grid

(nSolid1 x nSolid2)

Insert initial particles

t < total No

time

Yes

Initialize process variables

Allocate memory

and copy

Process variables - GPU copy

Calculate

Aggregation rates

Kernel launch command

Agg. kernel grid (nComp.,nSolids1 x nSolids2)

Calculate Breakage rates

Kernel launch command

Breakage kernel grid (nComp.,nSolids1 x nSolids2)

Calculate

Consolidation rates

Kernel launch command

Conso. kernel grid (nComp.,nSolids1 x nSolids2)

Assign values to process variables

Copy process variables

Calculate variable timestep

t = t + timestep

Calculate final d50 values

End

Figure 2: Workflow of the GPU code indicating data transfers and execution timeline of the code.

**Algorithm 2** GPU-based Parallel Population Balance Model

1: **procedure** PBM(*NC omp* ) *1> NC omp* is the number of compartments

2: Copy initial variables from CPU memory to GPU memory

3: GPU initial calculation kernel call from CPU *1>* Performed on GPU

4: Divide *NC omp* in *Nblocks*

5: Copy back initial values to the CPU RAM

6: **while** *t < tf inal* **do**

7: Copy time-dependent process variables from CPU to GPU RAM

8: GPU aggregation rate kernel call from CPU

9: Calculate *'!Ragg* for solid bins *s*1 ,*s*2 *1>* Performed on GPU

10: GPU aggregation rate kernel call from CPU

11: Calculate *'!Rbreakage* for solid bins *s*1 ,*s*2 *1>* Performed on GPU

12: Calculate *nparticles* using Euler’s method

13: Copy back process rate data back to the CPU RAM

14: Calculate *timestep* using *CFL* condition *1>* Performed on CPU

15: *tnew* = *t* + *timestep*

16: **end while**

17: Clear GPU memory

18: **end procedure**

300

301

302

303

304

305

306

307

same and the problem size is increased. In this study, the number of processing units were limited due to the architecture of the GPU and the CUDA C++ code developed did not utilize more than one GPU during execution. Thus, a soft scaling approach was preferred in such a scenario. The parallel performance of a code is usually measured in terms of on ratio of time taken to solve the run the simulations on one core to the time taken to run the simulation on N cores. It is depicted in Equation 17, where *t*1 is the time taken to the run the problem

on one core where as *tN* is the time taken to run the problem on N cores.

*t*1

*Speedup* =

*tN*

(17)

308

309

310

311

The problem size was varied by increasing the number of compartments in- side the PBM. This in turn increased the total number of calculations performed without increasing the amount of work that needed to be performed by each processing unit (core).

312

313

314

315

316

317

318

319

320

321

322

323

324

325

326

327

328

329

330

331

332

333

334

335

336

337

338

339

340

341

342

*4.1. Algorithm performance on a desktop GPU*

The desktop configuration used for these studies comprised of a Intel *i*7 *−*

7700*K* CPU clocked at 4*.*5 GHz with 32 GB DDR4 RAM and a NVIDIA Quadro *P* 4000 GPU. The NVIDIA Quadro GPU used had 1792 CUDA cores with 8 GB of GDDR5 RAM. CUDA version 9.0 paired with GCC 7.3 was used to the run the desktop GPU simulations with Ubuntu 18.04 operating system (OS).

Each GPU consisted of several streaming multi-processors(SM) which help distribute the problem to the 1792 cores inside the GPU. Once the calculations pass from the CPU to the GPU, the SMs take over and allocate work to the GPU in blocks of 32 threads each. This distribution was found to be one of the major bottlenecks especially when control was handed over from the CPU to the GPU. This leads to a significant time delay if the amount of calculation to be performed by the GPU is low and the control exchanges are more common. The number of solid bins for the 2 different types of solids used were 16, this meant that there was a maximum of 65*,* 536 calculations that needed to be performed for each time step for each compartment. While, the number of calculations increased to over 2 million per time step when the number of compartments was increased to 32. SMs divide these calculations in blocks of 32 threads and send it to the cores for calculations which accounts for some overheard time during the simulation. This overhead is present for each timestep, which can be compensated by the number of calculations running in parallel on the GPU. The PBM was run for 90 seconds which included 45 seconds of mixing and 45 seconds of liquid addition. The algorithm performance was tested by soft scaling the problem by changing the number of compartments from 1 and doubling them in each simulation until the number of compartments reached

32. Figure 3 shows the time taken these simulation. It can be seen that the amount of time taken for the simulation remains almost constant till the number of compartments reaches 8, followed by an increase in the time taken as the number of compartment are increased further to 32. According to parallelization procedure used the cores utilized to run the code is directly proportional to the number of compartments and the number of solid bins present in the problem.

343

344

345

346

347

348

349

350

The constant time is a result of the problem size being smaller than the Quadro *P* 4000’s 1792 CUDA core, i.e. the algorithm was not able to utilize all the CUDA cores till the compartment number was 8. Since the algorithm uses about

256 cores to simulate each compartment, the cores would not suffice once the compartment number reaches 16 and the SMs would have to wait to distribute the calculation to cores once initially allocated calculations are finished. This wait time leads to the increase in the time of simulations as seen in the case for

16 and 32 compartments.

240

220

200

180

Time(s)

160

140

120

100

1 2 4 8 16 32

Number of Compartments

Figure 3: Time taken to complete 90 seconds of PBM simulation with varying number of

compartments on NVIDIA Quadro P4000 GPU

351

352

353

354

355

356

The above argument was supported by the profile of the code that was obtained using NVIDIA’s inbuilt code profiler *nvprof* . Code profiling is an important step in algorithm development. The profiler results can be varied based on the options chosen to obtain the parameters being studied. In this case, API calls and GPU activity were exported to understand the performance bottlenecks and those sections of the code were rectified to improve the speed

357

358

359

360

361

362

363

364

365

366

367

368

369

370

371

372

373

374

375

376

of the algorithm. The profiler was executed for each case and it was observed that aggregation kernel calculations took the most time for execution followed by the breakage kernel. Consolidation kernel and other calculations comprised of less than 1% of execution time. The other parameter studied was the number of times each API was summoned by the code and the time spent. API calls in- cluded the synchronization of threads working inside the GPU device, memory allocation for arrays, etc. Each thread inside the GPU operates independently, thus all threads may not be at the same line of code at a given moment of time, thus some threads finish calculations before others. The time taken to synchro- nize these threads required the most amount of time during the execution of the code. When such an API is called by the code further execution of the code is paused until all the threads of the GPU are in the same line of code. This accounted for 99% of the total other API call time. This indicated that there were not many places where the code could have been optimized further since synchronization statements were only added before calculations where complete array of data was required. If further reduction in these statements was under- taken, it would lead to data loss and possibly incorrect final calculated particle size distribution. A comparison of times taken by each process in the simula- tions is shown in Figure 4. A similar distribution of times was observed for all

simulations on the GPU.

377

*4.2. Performance on GPUs compared to CPUs*

378

379

380

381

382

383

384

385

386

The NVIDIA Quadro P4000 GPU used had its cores at a base clock speed of

1202 MHz while the CPU cores had a base clock of 4000 MHz. The algorithm used to parallelize on the GPU did not permit the use of only one core of the GPU for simulation. Thus, a single MPI core CPU simulation was used as the baseline for all comparisons. Theoretically, it would take longer on a single core of the GPU to run a similar simulation than on a single GPU core

The CPU version of the parallel PBM was run on the desktop with the afore- mentioned configuration. This meant that the number of MPI cores available for the simulations was limited to 4. Soft scaling of the problem by changing

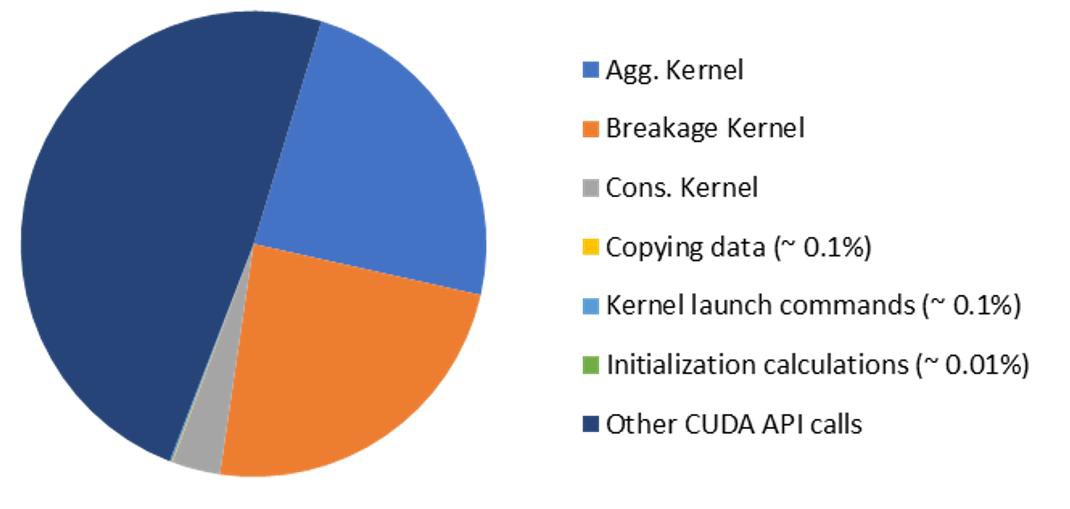


Figure 4: Distribution of times taken by different processes inside the GPU-parallelized PBM

387

388

389

390

391

392

393

394

395

396

397

398

399

400

the number of compartments was performed for this study. Figure 5 shows a comparison of the times taken by the simulation to run on 1, 2 & 4 MPI cores. The times indicate that with the increase in the number of cores the model took less time to complete calculations for the same number of compartments. It can also be seen that for the same number of MPI cores used in a soft scaling the amount of time increases with increase in the number of cores. This increase can be attributed to the increase in the number of calculations with addition of new compartments. There is a slight plateau in the times when 2 and 4 MPI cores were used for 8 and 16 compartments respectively. When a further analysis of the rates was for each compartment was undertaken it was observed that till the particles did not each the last few compartments of the granulator the rates were 0 thus reducing the compute time and leading to simulation times in a similar range.

Speedup is an important aspect that is considered to understand the scala-

103

1 MPI core

2 MPI core

4 MPI core

102

Time(s)

101

1 2 4 8 16

Number of Compartments

Figure 5: Time taken to complete 90 seconds of PBM simulation with varying number of

compartments on desktop CPU with varying number of MPI cores

401

402

403

404

405

406

407

408

409

410

411

412

413

414

415

bility and parallel performance of a code. Speedup for a code is directly propor- tional to the number of cores used for a simulation. In Figure 6, the speedup increases with the increase in the number of MPI cores. This increase in speedup can be attributed to the increase in the computation power. Another unusual trend observed in the case of 8, 16 and 32 number of compartments, the speedup is higher than 2 and 4 for 2 MPI and 4 MPI core simulations respectively. This phenomena is known as super linear speedup which occurs when the speedup is greater than the number of cores used. In rare cases like these speedup increases due to increase in cache memory and random access memory(RAM) available (Benzi and Damodaran, 2009). The simulations with the GPU parallel code showed an overall increase in the speedup as the number of compartments as seen in Figure 6. The speedup was low for compartment numbers 1 and 2 since the amount of time spent in communication in between the CPU and the GPU as well as the time taken by the SMs inside the GPU to distribute the problem had a larger contribution to the simulation time. The increase in num-

416

417

418

419

ber of compartments diminished this communication time effect as amount of calculations is significantly higher. The highest speeedup achieved for a GPU simulation was about 12*.*3, which means it took 12 times less time than a serial

CPU computation.

14

|  |  |
| --- | --- |
|  | 2 MPI cores |
|  | 4 MPI cores |
|  | GPU |

12

10

8

Speedup

6

4

2

0

1 2 4 8 16 32

Number of Compartments

Figure 6: Comparing speedup for CPU and GPU simulations to respective serial simulations

420

*4.3. Server level GPU code performance*

421

422

423

424

425

426

427

428

429

430

The high performance computing (HPC) device used to run the parallel PBM GPU code was present at Rutgers at the School of Engineering (SoE). The SoE HPC cluster was equipped with a NVIDIA Kepler K20 GPU. This GPU contains

2496 CUDA cores which are have a base clock of 706 MHz and only 5 GB of GDDR5 of memory. The Kepler series GPUs were a couple of generations older than the Pascal generation Quadro P400 used in desktop simulation studies. The clock speed and memory of the desktop GPU was higher than the one present on the HPC.

Time taken to complete the 90 second PBM simulation on the HPC’s GPU

are shown in Figure 7a. The time taken to run the PBM initially remains

431

432

433

434

435

436

437

438

439

440

441

442

constant upto 16 compartments, but a large increase in the time is observed for the simulation with 32 compartments. This increase could be attributed to the saturation of CUDA cores of the GPU and that SMs had to wait for the previous calculations to complete before the threads were assigned the remnant of calculations. A serial simulation was performed on the HPC and used as a baseline for speedup calculations. Speedup from the server GPUs are shown in Figure 7b. The increase in the speed of the simulation for these studies is lower than the desktop studies which could be directly connected to clock speeds of the CUDA cores. The server GPU cores were clocked at a lower frequency which meant the rate of calculations would decrease. One other reason for reduced speedup could be the older architecture of Kepler GPU which are slower in

floating point calculations (NVIDIA Corporation, 2016).

500 6

480

5

460

4

440

Time taken(s)

420 3

Speedup

400

2

380

1

360

340

1 2 4 8 16 32

Number of Compartments

0

1 2 4 8 16 32

Number of Compartments

(a)

(b)

Figure 7: (a)Time taken to run 90s PBM simulation on HPC’s Kepler K20 GPU (b)Speedup

achieved for the GPU simulation over the serial simulation on the HPC device

443

**5. Conclusions**

444

445

446

447

448

In the presented study, a PBM was developed using to run parallel on a GPU. The time of the simulations on the GPU were compared to similar ones on CPUs. In most of the cases it was observed that GPU simulations were faster than CPU simulations which had higher number of calculations. Simulations with low amounts of calculations were slower in case of GPUs, but such cases are

449

450

451

452

453

454

455

456

457

rare in PBMs. The GPU architecture also plays a major role in the simulation. This work also highlighted that a desktop PC could be used for computationally intensive simulations rather than a supercomputer or cluster when resources are limited. This work can be extended in the future by testing on newer GPUs from NVIDIA like the Volta and Turing platforms, which are more optimized for float point calculations than the Pascal platform GPU used in this study. The algorithm to parallelize the code can also be parallelized further to elimi- nate loops inside the kernels using dynamic parallelization supported by newer

versions of CUDA.

**References**

Almasi, G.S., Gottlieb, A., 1989. Highly Parallel Computing. Benjamin- Cummings Publishing Co., Inc., Redwood City, CA, USA.

Barrasso, D., Eppinger, T., Pereira, F.E., Aglave, R., Debus, K., Berming- ham, S.K., Ramachandran, R., 2015a. A multi-scale, mechanistic model of a wet granulation process using a novel bi-directional PBM–DEM coupling algorithm. Chemical Engineering Science 123, 500–513.

Barrasso, D., Ramachandran, R., 2015. Multi-scale modeling of granulation processes: bi-directional coupling of PBM with DEM via collision frequencies. Chemical Engineering Research and Design 93, 304–317.

Barrasso, D., Tamrakar, A., Ramachandran, R., 2015b. Model order reduction of a multi-scale pbm-dem description of a wet granulation process via ann. Procedia Engineering 102, 1295 – 1304.

Barrasso, D., Walia, S., Ramachandran, R., 2013. Multi-component population balance modeling of continuous granulation processes: a parametric study and comparison with experimental trends. Powder technology 241, 85–97.

Benzi, J., Damodaran, M., 2009. Parallel three dimensional direct simulation monte carlo for simulating micro flows, in: Parallel computational fluid dy-

namics 2007: implementations and experiences on large scale and grid com- puting. Springer Science & Business Media. volume 67, pp. 91–98.

Bettencourt, F.E., Chaturbedi, A., Ramachandran, R., 2017. Parallelization methods for efficient simulation of high dimensional population balance mod- els of granulation. Computers and Chemical Engineering 107, 158–170.

Chaturbedi, A., Bandi, C.K., Reddy, D., Pandey, P., Narang, A., Bindra, D., Tao, L., Zhao, J., Li, J., Hussain, M., Ramachandran, R., 2017. Compartment based population balance model development of a high shear wet granulation process via dry and wet binder addition. Chemical Engineering Research and Design 123, 187–200.

FDA, 2004. Guidance for Industry PAT — A Framework for Innovative Phar- maceutical Development, Manufacuring, and Quality Assurance. FDA official document , 16.

Gunawan, R., Fusman, I., Braatz, R.D., 2008. Parallel high-resolution finite volume simulation of particulate processes. AIChE journal 54, 1449–1458.

Immanuel, C.D., Doyle, F.J., 2005. Solution technique for a multi-dimensional population balance model describing granulation processes. Powder Technol- ogy 156, 213 – 225. Particle Technology Forum Special Issue.

Iveson, S.M., Litster, J.D., Hapgood, K., Ennis, B.J., 2001. Nucleation, growth and breakage phenomena in agitated wet granulation processes: a review. Powder technology 117, 3–39.

Kandrot, E., Sanders, J., 2011. Cuda By Example: An Introduction To General- Purpose Gpu Programming. Addison-Wesley Professional. URL: [https:](https://books.google.com/books?id=6mwanQAACAAJ)

[//books.google.com/books?id=6mwanQAACAAJ](https://books.google.com/books?id=6mwanQAACAAJ).

Keckler, S.W., Dally, W.J., Khailany, B., Garland, M., Glasco, D., 2011. Gpus and the future of parallel computing. IEEE Micro 31, 7–17.

NVIDIA Corporation, 2012. NVIDIA CUDA C Programming Guide. version

4.2 ed. NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050. URL: [https://developer.download.nvidia.com/compute/ DevZone/docs/html/C/doc/CUDA\_C\_Programming\_Guide.pdf](https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf).

NVIDIA Corporation, 2016. NVIDIA Tesla P100: The most advanced datacen- ter accelator ever built. Technical Report.

Prakash, A.V., Chaudhury, A., Ramachandran, R., 2013. Parallel simulation of population balance model-based particulate processes using multicore CPUs and GPUs. Modelling and Simulation in Engineering 2013, 2.

Ramachandran, R., Barton, P.I., 2010. Effective parameter estimation within a multi-dimensional population balance model framework. Chemical Engineer- ing Science 65, 4884–4893.

Ramachandran, R., Immanuel, C.D., Stepanek, F., Litster, J.D., Doyle, F.J.,

2009. A mechanistic model for breakage in population balances of granula- tion: Theoretical kernel development and experimental validation. Chemical Engineering Research and Design 87, 598–614.

Ramkrishna, D., Singh, M.R., 2014. Population balance modeling: current status and future prospects. Annual review of chemical and biomolecular engineering 5, 123–146.

Sampat, C., Bettencourt, F., Baranwal, Y., Paraskevakos, I., Chaturbedi, A., Karkala, S., Jha, S., Ramachandran, R., Ierapetritou, M., 2018. A parallel unidirectional coupled dem-pbm model for the efficient simulation of com- putationally intensive particulate process systems. Computers and Chemical Engineering 119, 128 – 142.

Santos, F.P., Senocak, I., Favero, J.L., Lage, P.L., 2013. Solution of the popu- lation balance equation using parallel adaptive cubature on gpus. Computers and Chemical Engineering 55, 61–70.

Sen, M., Barrasso, D., Singh, R., Ramachandran, R., 2014. A multi-scale hybrid

CFD-DEM-PBM description of a fluid-bed granulation process. Processes 2,

89–111.

Seville, J., Tu¨zu¨n, U., Clift, R., 2012. Processing of particulate solids. volume 9. Springer Science & Business Media.

Shi, Y., Green, W.H., Wong, H.W., Oluwole, O.O., 2012. Accelerating multi- dimensional combustion simulations using gpu and hybrid explicit/implicit ode integration. Combustion and Flame 159, 2388 – 2397.

Solihin, Y., 2015. Fundamentals of parallel multicore architecture. Chapman and Hall/CRC, New York, USA.

Szil´agyi, B., Nagy, Z.K., 2016. Graphical processing unit (gpu) acceleration for numerical solution of population balance models using high resolution finite volume algorithm. Computers and Chemical Engineering 91, 167 – 181.

Verkoeijen, D., Pouw, G.A., Meesters, G.M., Scarlett, B., 2002. Population balances for particulate processes—a volume approach. Chemical Engineering Science 57, 2287–2303.

Xu, Z., Zhao, H., Zheng, C., 2015. Accelerating population balance-monte carlo simulation for coagulation dynamics from the markov jump model, stochastic algorithm and gpu parallel computing. Journal of Computational Physics

281, 844 – 863.