VPIC

3D relativistic, electromagnetic Particle-In-Cell plasma simulation code

Description of VPIC

VPIC is a 3D relativistic, electromagnetic Particle-In-Cell plasma simulation code. The 3D grid or mesh is a structured Cartesian mesh with uniform grid spacing. Most of the computational work in a time step is done in a series of loops over either particles or grid cells. The average number of particles per cell can range from a few tens to a few thousands depending on the problem being solved.

VPIC was designed to use single precision floating point arithmetic to optimize the use of the available memory bandwidth. Both the particle data and the grid data are organized in an array of struct (AoS) storage format that is aligned along the appropriate word boundaries. Data is read and stored using SIMD vector loads and stores and is then transposed on the fly so it can be used in vector operations.

VPIC uses asynchronous MPI as the top level of parallelism, Pthreads or OpenMP as the middle level of parallelism and vectorization at the lowest level. The granularity of work assigned to a thread is large. To achieve vectorization, VPIC uses a light weight C++ vector wrapper class that wraps Intel intrinsic function implementations of basic math operations. There is an older 128 bit SSE implementation that uses a vector length of 4, a newer 256 bit AVX2 implementation that uses a vector length of 8 and a still newer 512 bit AVX512 implementation that uses a vector length of 16. There is also a reference implementation that does no explicit vectorization but instead leaves the vectorization task to the compiler.

VPIC uses a particle sort as an optimization to enhance cache locality by ordering the particles such that consecutive particles are located in the same cell. The particle sort is performed at a user specified frequency of time steps such that a heavy ion species which is less mobile is sorted less frequently relative to the sort frequency of electrons. VPIC also duplicates the number of copies of the particle accumulator arrays for current density for the multi-thread case to avoid the possibility of multiple threads attempting to update the current density of a cell at the same time. This strategy is reasonable for the case of a small number of threads per MPI rank for particle dominated problems where the number of particles per cell is large i.e. hundreds or thousands of particles per cell. It would not be reasonable for the GPU case where there might be thousands or tens of thousands of threads per MPI rank. For that case, an effective strategy would likely involve use of atomic operations and perhaps involve not sorting the particles or sorting them in a different manner. VPIC has two particle sort implementations, a legacy sort that is MPI parallel but thread serial which allows for either in-place or out-of-place sorts and a thread parallel sort which only supports out-of-place sorts. The particle sort implementation is chosen at compile time.

VPIC is written as a mixture of C and C++ code. The most compute intensive kernel in VPIC is the particle loop which gets called by the particle advance function.

Description of VPIC Archive Distribution

The files in this distribution of VPIC are organized in the following manner. The top level directory is vpic_crossroads. In vpic_crossroads are two directories, repos and vpic_project. Within the repos directory is a directory named git and within that are two directories, VPIC and vpic_project. The vpic

and vpic_project directories contain bare git repos of the VPIC source code and VPIC project infrastructure, respectively.

Within vpic_crossroads, the vpic_project directory is a working checkout of the vpic_project git directory which contains infrastructure for managing different branches of the VPIC git repository, building vpic on different machines of interest to LANL and running VPIC on these different machines. The vpic_project directory was cloned from the vpic_crossroads/repos/git/vpic_project bare git repository. Within the vpic_project directory are the following directories: bin, build, doc, runs and src.

The bin directory contains bash scripts used to build VPIC on several machines of interest to LANL. These scripts are organized in file pairs for each machine of interest. For instance, to generate benchmark results for the Crossroads RFP, scripts were used for Trinity i.e. makeVPIC_trinity and makeVPIC_trinity_cases. The top level script is makeVPIC_trinity_cases which configures several options required by the makeVPIC_trinity script and then calls the makeVPIC_trinity script once for each build case requested. Typically, the makeVPIC_trinity_cases script is copied to another file name and then manually customized by editing the new script. In this case, the new script name was makeVPIC_trinity_cases_int_hsw_lsort and was customized to build VPIC for the Trinity Haswell partition using the Intel compiler and the legacy MPI parallel but thread serial particle sorting capability. There is an additional custom top level script which builds VPIC for the Trinity Haswell partition using the Intel compiler and the MPI parallel and thread parallel particle sorting capability. This script is named makeVPIC_trinity_cases_int_hsw_tsort. One key difference between the legacy particle sort and the thread parallel sort is that the legacy particle sort can perform the sort in-place while the thread parallel sort only implements an out-of-place sort. Each of these two build scripts builds VPIC for two cases. One uses a non-vectorized implementation of the various VPIC compute kernels and might be a good place to start depending on the characteristics of the vendor architecture. The other build implements the VPIC compute kernels via explicit use of vector intrinsics. For the example case of Trinity Haswell, the 256 bit AVX2 SIMD intrinsics are targeted.

The build directory is used to contain the build products from the vpic build process. An "arch" directory is created for each build with a directory name that should be informative of the type of build. In the current build directory are four "arch" directories. They are:

TRINITY_CMPI_PTH_INT_OPT_V1_NONE_HSW_LSORT
TRINITY_CMPI_PTH_INT_OPT_V1_NONE_HSW_TSORT
TRINITY_CMPI_PTH_INT_OPT_V8_AVX2_HSW_LSORT
TRINITY_CMPI_PTH_INT_OPT_V8_AVX2_HSW_TSORT

For this set of examples, the "arch" names were composed by the top level build scripts i.e.

vpic_project/bin/makeVPIC_trinity_cases_int_hsw_lsort
vpic_project/bin/makeVPIC_trinity_cases_int_hsw_tsort

and passed to the lower level makeVPIC_trinity script. The lower level makeVPIC_trinity script creates the required "arch" directory and copies the VPIC source code to it for the specified git branch which in this case is the "crossroads" branch. Based on the features specified by the top level makeVPIC_trinity_cases_[hsw_lsort|hsw_tsort], the makeVPIC_trinity script configures various make variables and the cmake command line. The makeVPIC_trinity script then invokes the cmake command

line and the make command line to build the VPIC library and the VPIC executable script. Additionally, key features of the build environment are captured in a script named bashrc.modules which can be used in the scripts to build and run the executable. Results for the benchmark runs provided for the RFP were generated using the TRINITY_CMPI_PTH_INT_OPT_V8_AVX2_HSW_LSORT arch build. The VPIC executable script is a script that is used to compile the VPIC input file to produce a VPIC executable. The VPIC input deck is a C++ file. The VPIC executable script is located in the following location:

vpic_project/build/$ARCH/vpic/build/bin/vpic

The src directory is used to organize and hold clones of the vpic source code for various git branches of interest. For the RFP, only the "crossroads" branch is of interest. It contains the latest work to optimize VPIC on the Trinity ATS-1 machine located at LANL. The top level of the VPIC source directory is located in the following directory:

vpic_project/src/branches/crossroads/vpic

This top level directory is a working clone of the VPIC bare git repo located in vpic_crossroads/repos/git/vpic with the crossroads git branch checked out. The following script provides an example of how to create this working clone of the VPIC crossroads branch:

vpic_project/src/branches/clone_vpic_repos

The runs directory contains infrastructure for configuring selected input decks for VPIC and setting up run directories for building the vpic executable and performing various types of strong scaling and weak scaling runs on various machines of interest to LANL. Template files exist within the vpic_project/runs/VPIC_Test_Decks directory which can be used by the provided scripts to generate the actual C++ input file for a given problem. For the Crossroads benchmark runs on Trinity, the following input template is used:

vpic_project/runs/VPIC_Test_Decks/lpi/v_04/lpi.template.master.cxx

This input file template is used to generate an additional set of template files for the various combinations of processors and nodes that can be run for a given problem. These additional files are generated by executing the following script:

vpic_project/runs/VPIC_Test_Decks/lpi/v_04/make_input_deck_template_cases

This script has already been executed to generate these additional input file templates. The final step in generating an input file for vpic which can be compiled into an executable is to execute a machine specific script. This step is performed from a machine specific run directory. As an example, see the following run directory which was used for the single node Crossroads benchmark run.

vpic_project/runs/lanl/ats1/trinity/lpi/v_04/run_v8_avx2_nn_0001

If you change to this directory, you will see the results of building and running VPIC on Trinity for a single node run. This run directory contains 3 input files for VPIC to run 3 separate single node cases. The input files end with a .cxx file extension. These input files are used for running Laser Plasma Interaction (LPI)

problems for a problem size that uses most of the DDR memory on a node. These input files are for a single node with 32 or 64 MPI ranks per node and 1 or 2 threads per MPI rank. The soft links for bashrc.modules and VPIC were created by executing the make_links script as follows.

make_links TRINITY_CMPI_PTH_INT_OPT_V8_AVX2_HSW_LSORT

The make_links script takes a single argument which is one of the build arches in the vpic_project/build directory. The C++ input files were created by editing the make_input_decks_scaling_weak_hsw script in the run directory. The VPIC executables for each input file are then built by editing and then executing the build_vpic_int_scaling_weak_hsw script located in the run directory. The VPIC problems are then run by editing and then running one of the 3 run scripts located in the run directory.

Finally, the vpic_project directory has a doc directory and in this directory are two subdirectories, journal and presentations. The journal directory contains published journal articles which discuss the original design of vpic. The presentations directory contains various conference presentations from the last 4-5 years which mainly discuss results from attempts to optimize VPIC on various modern architectures.

How to Build and Run VPIC

In this section, a step-by-step process is provided for taking the provided six (6) VPIC distribution tar files, concatenating them and building and running the vpic Crossroads benchmark problem on a Trinity Haswell node.

a. Choose a location to concatenate, uncompress and untar the provided distribution tar files.

To reassemble the family of 6 split files into a single xzip tar file named vpic_crossroads.tar.xz, use the following command.

cat vpic_crossroads.tar.xz.* >& vpic_crossroads.tar.xz

To extract the files from vpic_crossroads.tar.xz, execute the following command.

xz -dc vpic_crossroads.tar.xz | tar xvf -

b. Change directories to the bin directory with the following command:

cd vpic_crossroads/vpic_project/bin

c. Copy the makeVPIC_trinity_cases to a new file name and edit it to make it identical to makeVPIC_trinity_cases_int_hsw_lsort with the following commands:

cp makeVPIC_trinity_cases makeVPIC_trinity_cases_int_hsw_lsort_v2

edit as required

diff makeVPIC_trinity_cases_int_hsw_lsort_v2 makeVPIC_trinity_cases_int_hsw_lsort

d. Change directories and execute the new build script just created.

cd ..

bin/makeVPIC_trinity_cases_int_hsw_lsort_v2 >& makeVPIC_trinity_cases_int_hsw_lsort_v2.log

e. Check to see if the build completed correctly.

cd build

find . -name libvpic.a

There should be a libvpic.a for each ARCH that was built and present in the vpic_project/build directory. Also, inspect the build log file for any errors or warnings. For this build, the build log file would be the following file:

vpic_project/build/$ARCH/vpic/makeVPIC_trinity.log

Also, inspect bashrc.modules to see if it seems reasonable. It is located at:

vpic_project/build/$ARCH/vpic/bashrc.modules

f. Change directories to the machine and problem specific directory. For this example change directories to the following directory and create a new run directory:

cd vpic_project/runs/lanl/ats1/trinity/lpi/v_04

mkdir run_v8_avx2_nn_0001_test

cd run_v8_avx2_nn_0001_test

g. Now, copy files from the machine and problem specific bin directory located in the parent directory and remove files specific to either strong scaling runs or knl runs because this run will be a weak scaling haswell run.

cp ../bin/* .

rm *strong*

rm *knl*

h. Now, make soft links to the vpic and bashrc.modules scripts for the desired build arch as follows:

./make_links TRINITY_CMPI_PTH_INT_OPT_V8_AVX2_HSW_LSORT

i. Now, edit make_input_decks_scaling_weak_hsw and make it identical to the make_input_decks_scaling_weak_hsw script in the ../run_v8_avx2_nn_0001 directory.

j. Next, edit build_vpic_int_scaling_weak_hsw and make it identical to the build_vpic_int_scaling_weak_hsw script in the ../run_v8_avx2_nn_0001 directory.

k. Next, edit run_vpic_int_scaling_weak_hsw_mpi and make it identical to the run_vpic_int_scaling_weak_hsw_mpi script in the ../run_v8_avx2_nn_0001 directory.

l. Now, make the input C++ files and compile them using the following commands:

./make_input_decks_scaling_weak_hsw

./build_vpic_int_scaling_weak_hsw

m. At this point, you should have a .Linux executable file for each .cxx file in the run directory. Run the executable file with a command like the following:

sbatch --nodes=1 --time=2:00:00 ./run_vpic_int_scaling_weak_hsw_mpi

Note that the LANL Trinity machine uses Slurm as the batch scheduler. This run should take about 40 minutes. A successful run should terminate with a "normal exit" line at the end of the output log file.

Crossroads Benchmark Problems

a. Small problem

1 node, 272 x 64 x 64 mesh, 1024 particles/cell/species, 1000 steps, 3 species
directory: vpic_project/runs/lanl/ats1/trinity/lpi/v_04/run_v8_avx2_nn_0001
input file: lpi_ddr_nn_0001_nppn_064_ntpp_001.cxx
Total Cycle Time: 2349.63 seconds
FOM: 1.457e9 particles/second =
    272 x 64 x 64 x 1024 x 1000 x 3 / 2349.63

b. Medium problem

64 nodes, 1088 x 256 x 256 mesh, 1024 particles/cell/species, 1000 steps, 3 species
directory: vpic_project/runs/lanl/ats1/trinity/lpi/v_04/run_v8_avx2_nn_0064
input file: lpi_ddr_nn_0064_nppn_064_ntpp_001.cxx
Total Cycle Time: 2370.51 seconds
FOM: 92.403e9 particles/second =
    1088 x 256 x 256 x 1024 x 1000 x 3 / 2370.51

c. Large problem

4096 nodes, 4352 x 1024 x 1024 mesh, 1024 particles/cell/species, 1000 steps, 3 species

directory: vpic_project/runs/lanl/ats1/trinity/lpi/v_04/run_v8_avx2_nn_4096
input file: lpi_ddr_nn_4096_nppn_064_ntpp_001.cxx
Total Cycle Time: 2380.58 seconds
FOM: 5888.806e9 particles/second =
4352 x 1024 x 1024 x 1024 x 1000 x 3 / 2380.58

Correctness

If VPIC runs successfully to completion, the log file which collects the output of standard out will end with the string "normal exit". Examples of log files for successful runs are contained in the directories above for the small and medium problems.

VPIC Specific Run Rules

a. Section 4 above provides the computational mesh for each of the three problem sizes and this needs to be preserved. In the C++ input files, the mesh dimensions are contained in the variables nx, ny and nz for the x, y and z coordinates respectively. An MPI topology description is also created in the input file to be used for the domain decomposition. The variables topology_x, topology_y and topology_z describe the number of MPI ranks in the x, y and z coordinates respectively. The mesh coordinates, nx, ny and nz are required to be an integer multiple of topology_x, topology_y and topology_z respectively. The dimensions of each of the MPI domains can then be described as (nx/topology_x) x (ny/topology_y) x (nz/topology_z). The problems in Section 4 each have 1024 particles/cell/species which is representative of actual production problems for particle dominated problems.

In order to provide more flexibility to the vendor to meet the VPIC mesh constraints without resorting to a large number of threads per MPI rank, the following additional constraints are specified for the mesh of the large problem.

nx_base = 4352
ny_base = 1024
nz_base = 1024

The values of nx, ny and nz can be chosen to fall in the following ranges.

3916 <= nx <= 4788
921 <= ny <= 1127
921 <= ny <= 1127

The product of nx x ny x nz must be greater than or equal to the product of
nx_base x ny_base x nz_base.

This ensures that the size of a customized problem is still greater than or equal to that of the reference large problem for which an FOM is provided as measured on the Trinity Haswell partition.

Following are some examples of how the above constraints may be met with a custom mesh.

Case 1: Embed a factor of 5 or 7 in mesh, version 1.

nx_base = 4352 = 17 x 4 x 8 x 8
nx = 4760 = 17 x 5 x 7 x 8 < 4788 and > nx_base
ny = 1024 = ny_base
nz = 1024 = nz_base

Case 2: Embed a factor of 5 or 7 in mesh, version 2.

nx_base = 4352 = 17 x 2 x 8 x 16
nx = 4480 = 7 x 5 x 8 x 16 < 4788 and > nx_base
ny = 1024 = ny_base
nz = 1024 = nz_base

Case 3: Embed a factor of 3 or 11 in mesh.

nx_base = 4352 = 17 x 8 x 16 x 2
nx = 4488 = 17 x 8 x 11 x 3 < 4788 and > nx_base
ny = 1024 = ny_base
nz = 1024 = nz_base

Case 4: Embed a factor of 18 = 2 x 3 x 3 in mesh.

nx_base = 4352 = 17 x 4 x 8 x 8
nx = 4608 = 18 x 4 x 8 x 8 < 4788 and > nx_base
ny = 1024 = ny_base
nz = 1024 = nz_base

Case 5: Embed a factor of 19 in mesh, version 1.

nx_base = 4352 = 17 x 4 x 8 x 8
nx = 4370 = (int(4352 / 19) + 1) x 19
= 4370 = 19 x 5 x 2 x 23 < 4788 and > nx_base
ny = 1024 = ny_base
nz = 1024 = nz_base

Case 6: Embed a factor of 19 in mesh, version 2.

nx_base = 4352 = 17 x 4 x 8 x 8
nx = 4560 = int((4352 + 4788) / 2 / 19) x 19
= 4560 = 19 x 5 x 3 x 16 < 4788 and > nx_base
ny = 1024 = ny_base
nz = 1024 = nz_base

b. Most of the computational work of VPIC is performed in loops over either particles or cells. There is no requirement to process either the particles or cells in any particular order.

c. A particle sort is used in the benchmark runs for the Crossroads RFP. The particle sort is an optimization and does not need to be performed. It is also permissible to sort the particles in other ways if that is beneficial to a different architecture.

d. The algorithms used in VPIC are carefully designed to yield certain desired numerical properties including maximizing the number of significant bits of accuracy in the calculation so that a single precision representation of the data can be used. An example of this is defining the coordinate of a particle in terms of a cell index plus a local coordinate within the cell. These algorithms and their numerical properties must be preserved by any modifications of VPIC to port it to another architecture and optimize it. In general, when porting to another architecture, it is permissible to change the layout of data in memory by changing the current array of structs to an alternative layout such as struct of arrays or array of struct of arrays. It is also permissible to sort the particles for processing in different ways or not at all. It is permissible to change the order in which particles or cells are processed. It is permissible to use alternative approaches to eliminate data collisions in the current density accumulation such as atomics or some data coloring scheme. It is not permissible to make changes to the algorithms which change the numerical properties such as the order of accuracy of the algorithms. It is not permissible to alter the interpolation schemes used to interpolate between particles and the grid. It is not permissible to change the algorithms for advancing particles and fields in time in a way that changes their numerical properties.

e. Within the C++ input files, it is not permitted to change any of the physics parameters or problem geometry parameters for the required problems in Section 4. The required problems were generated from the following input file template:

vpic_project/runs/VPIC_Test_Decks/lpi/v_04/lpi.template.master.cxx

This file contains a number of strings that begin with the prefix "REPLACE_". These strings get replaced by appropriate values using sed by one of two scripts that were described in Section 2 above. It is permissible to modify the values which correspond to REPLACE_*_sort_* since these control the sort of the particles. Replacing the REPLACE_*_sort_interval string with a negative integer will turn off the particle sort. If it is useful to sort the particles, the sort interval can be tuned to minimize the run time.

It is also permissible to change the shape of the domain decomposition consistent with the limitations imposed by the grid dimensions for the required problems.

VPIC Figure of Merit (FOM)

The FOM for a VPIC run is the number of particles/second which are advanced in time. This can be calculated as follows:

FOM = nx x ny x nz x 1024 x 1000 x 3 / total_cycle_time

The total_cycle_time is reported in seconds in the log file and may be found by searching for the string "*** Done". See examples of the FOM in Section 4 above.

Reporting

For the electronic submission, include the VPIC source and build products from the vpic_project/build/ARCH directory and any scripts used to build on the target platform. Include the C++ input source file and all output files from the runs. Include any run scripts used to perform the run.

Information which may be useful for porting to new architectures

a. There is a non-vectorized, reference implementation of each of the vpic compute kernels. For example, see the advance_p_pipeline function in the advance_p.cc source file located in the following directory:

vpic_project/src/branches/crossroads/vpic/src/species_advance/standard

The non-vectorized, reference implementation is a good place to start when porting to a new architecture, especially if the new architecture differs in significant ways from a traditional cache-based architecture with SIMD vectors. The reference implementation can also be used to verify the correctness of a new implementation.

b. The vectorized versions of VPIC use a light weight C++ wrapper class which is implemented using vector intrinsic functions for commonly available SIMD vector types. There is a full implementation for 128 bit and 256 bit SIMD vectors and a partial implementation for 512 bit SIMD vectors. The vector implementations are referred to as v4, v8 and v16 because of the number of float sized elements that will fit within the SIMD vector. For each of the vector implementations there are two portable implementations of the light weight vector functions, for instance, v8_portable_v0.h and v8_portable_v1.h for the v8 class. The portable implementations use basic C++ implementations in either unrolled loop format or short loop format. The portable implementations are useful for debugging and verifying correctness. They are also useful as a starting point for implementing a new intrinsics implementation for a different architecture. One of the portable implementations can be copied to the header file for a new intrinsics implementation and then the different portable implementations can be replaced incrementally with an intrinsics implementation and correctness can be verified incrementally.

c. Key data structures can be found in the following two files:

vpic_project/src/branches/crossroads/vpic/src/species_advance/species_advance_aos.h

for particle data structures and

vpic_project/src/branches/crossroads/vpic/src/sf_interface/sf_interface.h

for grid or mesh data structures. The data layout of these data structures may be modified to map better to the desired architecture as long as the actual algorithm is not changed.

d. For the required problems of Section 4, the memory footprint is dominated by the particle data structures. The memory used by the particle data structures can be estimated as

particle_memory = 32 bytes/particle *

nx * ny * nz *
nppc *
number_of_species *
REPLACE_max_local_np_scale

where nx*ny*nz is the total number of cells, nppc is the number of particles per cell and REPLACE_max_local_np_scale=1.1 is a scale factor to allow over allocation of memory for particles to account for non-uniformity of the particle distribution from either the initial conditions or the problem evolution. Thus, the total number of particles/species is nx*ny*nz*nppc. The required problems have number_of_species=3.

e. VPIC generates inclusive timing statistics for several functions and these statistics are printed to standard out for MPI rank 0. Since VPIC uses an asynchronous MPI implementation, if there is any load imbalance from something like a slow node, the load imbalance time will show up in the timing data for the boundary_p function which acts like a barrier because of the MPI calls in this function.