# Vectorizing Device Model Evaluation in Ngspice circuit simulator

Florian Ballenegger, Anamosic Ballenegger Design

July 2020

## Abstract

A method improving the execution speed of electrical circuit simulation using vector processing is proposed. The BSIM3V32 semi-conductor device model for the open-source Ngspice simulator has been re-written for evaluating multiple device instances of the same model at once using Single Instruction Multiple Data (SIMD) processor instructions. While parallel evaluation of device model was already available using multiprocessing, the proposed method can achieve the same speed-up using less processor resources, thus allowing to do more parallel independant simulations for statistical analysis.

## 1 Introduction

Circuit simulators are today a key tool for the successfull design and verification of modern integrated circuits (IC). Ngspice is an open-source simulator which has evolved from the original SPICE3 code developped at the University of California (US). Anybody can test new features by enhancing the open-source code.

With the increasing complexity of IC designs, the ability to simulate a circuit in many conditions in less time is more and more important. While the clock frequency of modern processors has not significantly increased recently, their architecture now offers several forms of parallelism which can be exploited to drastically increase the execution speed of a program. The two main forms of parallelism are:

- *Multiprocessing* or *task-parallelism.* Processors commonly includes several cores on the same chip, each core can execute a different task.

- *Vector processing* or *data-parallelism.* Specific instructions can process multiple data at the same time. The same task can be made on several data set simultaneously, with typical vector length of 4 double precision floating point numbers (AVX2) or 8 (AVX-512).

The semi-conductor device model evaluation is the task taking the most execution time of the simulation for medium-sized circuits [2]. Matrix solving dominates only with circuits having huge number of nodes, as it is often the case when simulating circuit with precise extracted interconnect parasitics.

While Ngspice does support parallel evaluation of device models using multiprocessing through the OpenMP API [6], it can be argued that in some important use cases this method does not bring any benefit, especially when corners or statistical MonteCarlo analysis is to be performed. As the multiprocessing resources will already be used by running several independant simulations simultaneously on different cores, using those same resources for parallel device model evaluation does not achieve any global speed-up.

In constrast, vector processing is a cheaper resource and does not compete with multiprocessing when several simulations are run for statistical analysis. Furthermore, vector processing and multiprocessing can be combined when only one simulation run is desired, achieving even greater speed-up.

In this work we implemented vectorized evaluation of MOSFET devices using the BSIM3V32 model in the Ngspice simulator. Compared to previously published works [1, 7, 8] which either operate on special hardware or within a totally new simulator architecture, our method easly integrates into a well established open-source simulator and runs on common hardware.

## 2 Implementation

### 2.1 Overview

In general, we can distinguish four methods for using vector processing in a program code:

- *Automatic compiler vectorization*
  The code is written in a way that the compiler can automatically vectorize, either knowing how the compiler works or using the OpenMP directives `omp parallel simd`.

- *Use of vectorized library*
  All calculation are made trough a library which is carefully implemented for using the vector processing resources.

- *Compiler vector extension*
  GCC and clang compilers both support vector extension for the C language. With those extensions the same operators used in sequential code (`+`, `-`, `*`, `/`, `&`, etc) are re-interprated to operate on vector data.

- *SIMD Instrinsics.*
  The *intrinsics* are like compiler built-in functions that directly map SIMD instructions for a specific

target processor architecture. They can be used from the C language on C-declared variables[1].

Vectorizing condititional branches is the main challenge. If the branch to execute depends on the data processed, different arms could need to be executed when processing several data in a vector. The common solution is to compute all arms and a mask vector capturing the condition. Then the vectors computed in the different arms are combined using the condition mask by a so-called *blend* operation [4].

It has been found that the GCC compiler[2] is unable to automatically vectorize the BSIM3V32 model code due to many conditional branches and function calls present, even if using OpenMP `parallel simd` constructs with detailed directives. Thus we decided to write explicit vector code using the compiler vector extension, which allows for a more portable code compared to using intrinsics, specific to e.g. the x86_64 architecture.

## 2.2 Instance grouping

We first indentify the data and parameters which are *uniform*, i.e. which are the same for all device instances evaluated in one vector. In order to minimize the number of conditional branches to be transformed into masked blending recombination, the instances are first grouped by similarity based on the `W`, `L`, `geo`, `nqsMod` and `off` parameters.

## 2.3 Source code transformation

The BSIM3V32 code has more than three thousand lines. To manually transform this chunk of code would be too tedious and error-prone. We decided to write a tool called `simdify` to automatically perform the required transformations. `simdify` is written in python using the widely available C parser *pycparser* to generate an Abstruct Syntax Tree (AST) of the original code. This tree is then analyzed and transformed by the tool, before being written back into C language, using the same *pycparser* module. The number of elements packed into one SIMD vector is configurable on invokation of the tool, and is denoted `NSIMD`.

In the following code examples, `NSIMD`=4. The original model code snippets are denoted with a light red background, and `here` refers to the processed instance data structure. The transformed SIMD model code snippets are denoted with a light green background, and `heres[NSIMD]` refer to a vector array of all processed instances in one SIMD evaluation.

The tool performs the operations enumerated below:

1. For every expression, recursively find if it depends on uniform data only. Attach this information to the assigned variable when an assignment operator is encountered.

[1]Compared to assembly, this avoids the need to manually allocates register and to manage the stack.

[2]and probably other compilers too

2. Alter declaration of variables for using vector type for all non-uniform variables.

```
double dT1_dVg;
```

```
Vec4d dT1_dVg;
```

3. Indentify conditional branches which depend on non-uniform data. Transform those branches into a vectorized version using masked blending [3].

```
if (T0 >= - 0.5)
{ T1 = 1.0 + T0;
  T2 = pParam->BSIM3v32dvt2;
}
else
{ T4 = 1.0 / (3.0 + 8.0 * T0);
  T1 = (1.0 + 3.0 * T0) * T4;
  T2 = pParam->BSIM3v32dvt2 * T4 * T4;
}
```

```
if (1)
{
  Vec4m mask0 = T0 >= (-0.5);
  Vec4m mask_true0 = mask0;
  Vec4m mask_false0 = ~mask0;
  {
    T1 = vec4_blend(T1, 1.0 + T0, mask_true0);
    T2 = vec4_blend(T2, vec4_SIMDTOVECTOR(pParam
    ->BSIM3v32dvt2w), mask_true0);
  }
  {
    T4 = vec4_blend(T4, 1.0 / (3.0 + (8.0 * T0)),
     mask_false0);
    T1 = vec4_blend(T1, (1.0 + (3.0 * T0)) * T4,
    mask_false0);
    T2 = vec4_blend(T2, (pParam->BSIM3v32dvt2w *
    T4) * T4, mask_false0);
  }
}
```

4. When a vector variable was assigned to a constant, replace this scalar constant with a vector constant[4].

```
dQac0_dVd = 0;
```

```
dQac0_dVd = (Vec4d ){0, 0, 0, 0};
```

5. When some non-uniform instance data was loaded from memory, load data from several instances into a single vector (*gather* operation).

```
V3 = here->BSIM3v32vfbzb
     - Vgs_eff + VbseffCV - DELTA_3;
```

```
V3 = (( ( (Vec4d ){
 heres[0]->BSIM3v32vfbzb,
 heres[1]->BSIM3v32vfbzb,
 heres[2]->BSIM3v32vfbzb,
 heres[3]->BSIM3v32vfbzb})
 - Vgs_eff) + VbseffCV) - DELTA_3;
```

6. When some uniform instance data was loaded from memory, just load the data for the first instance into a scalar value. This can happend because the devices were grouped for sharing same values for this parameter.

```
if (here->BSIM3v32nqsMod)
```

```
if (heres[0]->BSIM3v32nqsMod)
```

7. When some instance data was written to memory, write each element of the vector to each instance data (*scatter* operation).

8. The above instance data read and write transformations also apply to circuit state read or write which are recognized by the tool.

[3]Unlimited number of imbricated conditional branches are handled, however with some impact on the performance of the transformed code.

[4]The compiler allows to combine scalars and vectors in operators found in expressions, but not in assignments.

```
here->BSIM3v32cgsb = -(Cgg + Cgd + Cgb);
```

```
{
  Vec4d val = -((Cgg + Cgd) + Cgb);
  heres[0]->BSIM3v32cgsb = val[0];
  heres[1]->BSIM3v32cgsb = val[1];
  heres[2]->BSIM3v32cgsb = val[2];
  heres[3]->BSIM3v32cgsb = val[3];
}
```

9. For function calls, the function name is prefixed for indicating that a vector version of the function need to be called. The vector version of the function needs then to be linked to an existing equivalent vector function from a library, or to be written by hand.

```
ExpVgst = exp(T0);
```

```
ExpVgst = vec4_exp(T0);
```

10. For function calls interacting with the simulator internals, the same call is just made sequentially with the data for each instance in the vector separately.

```
error = NIintegrate(ckt, &geq, &ceq, 0.0, here->
    BSIM3v32qb);
```

```
static inline int
vec4_NIintegrate(CKTcircuit* ckt, double* geq,
    double *ceq, double zero, Vec4m chargestate)
{
  int error;
  for(int idx=0;idx<NSIMD;idx++)
  {
    error = NIintegrate(ckt,geq,ceq,zero,
    chargestate[idx]);
    if(error) return error;
  }
  return error;
}

error = vec4_NIintegrate(ckt, &geq, &ceq, 0.0,
  (Vec4m ){
  heres[0]->BSIM3v32qb, heres[1]->BSIM3v32qb,
  heres[2]->BSIM3v32qb, heres[3]->BSIM3v32qb});
```

## 2.4 Manual modifications

Some modifications in the original code have been made manually.

### 2.4.1 Reduction

In one place in the code, the number of non-converged devices is added into a global counter:

```
if (Check==1) ckt->CKTnoncon++;
```

This has to be replaced by a count of the non-converged devices in the processed vector:

```
ckt->CKTnoncon += SIMDCOUNT(Check)'
```

where the function SIMDCOUNT performs a horizontal sum reduction on the `Check` vector.

### 2.4.2 Optimization

Math functions like `exp` and `log` are time consuming. Where the same exp or log computation was made in diffrent arms of a conditional branch, it is more efficient to move and precompute this expression outside the conditional branch. This way only one expensive computation is made except of two or more.

### 2.4.3 Vector Mathematical Functions

Vector implementation of 5 mathematical functions must be provided: `exp`, `log`, `sqrt`, `MAX` and `fabs`, all function being prefixed with `vecN_` where `N=NSIMD`. The blending operation `vecN_blend` must also be provided.

Some of those functions are available as x86_64 intrinsics[3]. While using intrinsics leads for sure to a loss of portability, the speed advantage is noticeable. It would be easy to change some macro definition to support a more portable solution.

GCC compiler comes with a vector mathematical library called libmvec[5], which conveniently implements a vector version of `exp` and `log` functions very efficiently[6].

Another option would be to use an open-source vector mathematical library written in C as in [5].

The following mapping is used when `NSIMD`=4:

| Function | Mapped to | Type |
|---|---|---|
| `vec4_sqrt` | `_mm256_sqrt_pd` | intrinsic |
| `vec4_MAX` | `_mm256_max_pd` | intrinsic |
| `vec4_blend` | `_mm256_blendv_pd` | intrinsic |
| `vec4_exp` | `_ZGVdN4v_exp` | libmvec |
| `vec4_log` | `_ZGVdN4v_log` | libmvec |
| `vecN_fabs` | `vecN_blend(x,-x,x<0)` | equival. |

## 2.5 Bypass

The BSIM3V32 model supports a *bypass* mode which applies when the terminal voltages of a device do not change. In this case it is not required to evaluate the model again and the calculations are just skipped. However in a SIMD environment, the control flow must be the same for the whole data set in the SIMD vector, thus it is not possible to apply bypass for only a few instances in the processed vector.

As work-around solution, the following mechanism is used. First the original sequential model code is entered up to the point where it is decided if bypass has to be applied. If bypass occurs, the sequential model just completes (skipping all calculations) and the device is marked as completed. In the other case, the sequential model code stops, stores 7 intermediate data calculated so far, and returns. The instances for which the sequential model has not yet completed due to bypassing are then collected in groups of `NSIMD` devices to be evaluated by the SIMD model code. The latter starts where the sequential code has stopped, loading the 7 intermediate data previously stored.

The remaining devices which do not fit into a full vector of `NSIMD` devices are eventually evaluated by the original sequential code.

# 3 Results

Both the original Ngspice software version 32 and the modified code was compiled with GCC[7] version 9.3.0

[5]The same vector math library is also available when using the clang compiler with vector extensions.

[6]with however a small loss of accuracy

[7]compilation with clang yield similar execution speed

in different configurations as summarized in table 1. The `-march=native` flag was also specified.

All versions are executed on a computer powered by an i7-6700 CPU running at 3.40GHz and featuring 4 physical cores, each core including the AVX2 vector processing unit. The operating system is linux Ubuntu 16.04. The version compiled with OpenMP multiprocessing was executed with 4 threads.

The test circuit is a ring oscillator with 128 stages, simulated with transient analysis for 100ns (about 10 oscillator periods). The transistors use a BSIM3V3 model from the industry for a 0.18 um technology.

For the test case *T1*, one simulation at typical conditions is launched, while for test case *T16*, 16 simulations with different power supply and temperature conditions are launched in parallel[8].

| | Original Ngspice | | Modified | |
|---|---|---|---|---|
| | Normal | MP | SIMD | SIMDMP |
| optim | O3 | O3 | O3 | O3 |
| OpenMP | no | yes | no | yes |
| T1 [s] | 10.54 | 5.4 | 5.53 | 3.64 |
| speed-up | 1 | 1.95 | 1.9 | **2.9** |
| T16 [s] | 38.0 | 84.9 | 22.33 | 65.36 |
| speed-up | 1 | 0.45 | **1.7** | 0.58 |
| T16/T1 | 3.6 | 15.7 | 4.04 | 17.7 |

Table 1: Execution speed comparison.

For test *T1*, the proposed vector processing achieves a speed-up of 1.9x, while multiprocessing is at 1.95x. Combining vector processing and multiprocessing is even faster with an 2.9x speed-up. This shows that the two approaches are using complementary computing resources which do not compete.

For test *T16* however, only the vector processing approach is faster at a decent 1.7x speed-up, while the use of multiprocessing is slower and even counter-productive. This shows that for a batch of multiple simulations, it is more efficient to use the processor cores to run multiple independant simulations in parallel than for multiprocessing inside each simulation.

The batch of 16 simulations of *T16* completes in only 4.04 more time than the single simulation of *T1*, thanks to the power of the 4 processor cores which by this way proves to still be fully available when using vector processing in each core[9].

## 4 Conclusions

Our work shows that vectorization of device model evaluation in a circuit simulation is possible and efficient when running on common CPUs found in modern desktop and server computers.

[8]For some reason, running multiple Ngspice simulations with OpenMP at the same time was performing extremely slowly, thus we instead run the 16 simulations sequentially.

[9]T16/T1=**4.04** is not exactly 16 sims / 4 cores = **4** because each simulation in the batch has different conditions and does not takes exactly the same time. Also heat production under intensive use does slow down the processor a bit.

By using SIMD instructions, only one processor core will be loaded by one simulation, which allows to perform statistical analysis by running multiple simulations in parallel on all processor cores efficiently.

For when only one simulation is required, further acceleration is achieved by combining the proposed vectorization with OpenMP multiprocessing.

In futur work and experimentations, the performance using vectors with 8 elements should be investigated, either on a computer with an AVX-512 unit, or by packing 8 single precision float in an AVX2 unit.

Only the BSIM3V32 device model was modified to use vector processing. Other device models would of course also benefit from the proposed method. In particular interest would be the EKV model [9], as the calculations in this symmetric model are more linear with fewer conditional branches and could be vectorized more efficiently.

For compact models written in Verilog-A, a promising approach would be to perform code vectorization during model compilation, instead of transforming the C code.

The source code of the modified BSIM3V3 model is available at `https://www.anamosic.com/pages/ngspice.html`.

## References

[1] A. Vladimirescu, *LSI Circuit Simulation on Vector Computers*, Memorandum No. UCB/ERL M82/75, 1982.

[2] F. Lannutti, P. Nenzi and M. Olivieri, *KLU sparse direct linear solver implementation into NGSPICE*, Proceedings of the 19th International Conference Mixed Design of Integrated Circuits and Systems - MIXDES 2012, Warsaw, 2012, pp. 69-73.

[3] Intel Intrinsics Guide, `https://software.intel.com/sites/landingpage/IntrinsicsGuide`

[4] Wende F., Noack M., Steinke T., Klemm M., Newburn C.J., Zitzlsberger G. (2016) *Portable SIMD Performance with OpenMP* 4.x Compiler Directives.* In: Dutot PF., Trystram D. (eds) Euro-Par 2016: Parallel Processing. Euro-Par 2016. Lecture Notes in Computer Science, vol 9833. Springer, Cham

[5] Christoph Lauter, *A new open-source SIMD vector libm fully implemented with high-level scalar C*, 2016 50th Asilomar Conference on Signals, Systems and Computers, Nov 2016, Pacific Grove, US. pp.407 - 411, 10.1109/ACSSC.2016.786907. hal-01511131

[6] R. Perng, T. Weng, and K. Li, *On performance enhancement of circuit simulation using multithreaded techniques,* in Computational Science and Engineering, 2009. CSE'09. International Conference on, vol. 1. IEEE, 2009, pp. 158–165.

[7] H. Peng and C. Cheng, *Parallel transistor level full-chip circuit simulation,* 2009 Design, Automation & Test in Europe Conference & Exhibition, Nice, 2009, pp. 304-307, doi: 10.1109/DATE.2009.5090677.

[8] K. C. A. Lam and M. Zwolinski, *"Circuit simulation using state space equations,"* Proceedings of the 2013 9th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME), Villach, 2013, pp. 177-180, doi: 10.1109/PRIME.2013.6603135.

[9] Enz, C. C.; Krummenacher, F.; Vittoz, E.A. (1995), *An Analytical MOS Transistor Model Valid in All Regions of Operation and Dedicated to Low-Voltage and Low-Current Applications*, Analog Integrated Circuits and Signal Processing Journal on Low-Voltage and Low-Power Design 8: 83–114, July 1995, doi:10.1007/BF01239381