Simple, Parallel Virtual Machines for Extreme Computations

arXiv: 1411.3834

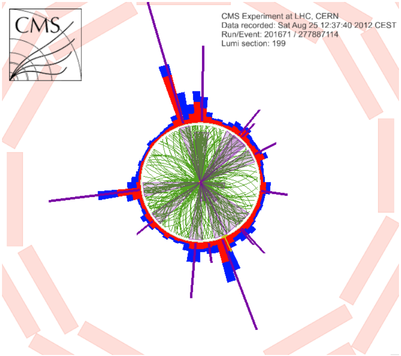
Bijan Chokoufe Nejad

Theory Group, DESY Hamburg

2nd International WHIZARD Forum

March 18th, 2015

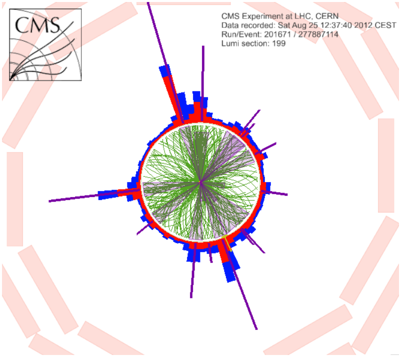
Motivation for Tree Level Matrix Elements (MEs)


CMS
CMS Experiment at LHC, CERN
Data recorded: Sat Aug 25 12:37:40 2012 CEST
Run/Event: 201671 / 277887114
Lumi section: 199

13 jet event with $p_{\mathrm{T,min}} = 50\,\mathrm{GeV}$ at $\sqrt{s} = 8\,\mathrm{TeV}$ of $12\,\mathrm{fb}^{-1}$ set (anti-k_{T}; $R = 0.5$)
[CMS–EXO–12–009]

- Need MEs for very high multiplicities to test the SM and its possible extensions
- MEs at tree level are part of every simulated event:
  - Correction of the shower to account for interferences (Merging)
  - Real emissions as part of the higher order computations: N^{k}LO implies up to k additional particles
- Automatized, efficient ME generators work in principle for every multiplicity: ALPHA [Caravaglios, Moretti 1995], HELAC [Kanaki and Papadopoulos 2000], O'MEGA [Moretti, Ohl, and Reuter 2001], COMIX [Gleisberg and Höche 2008], . . . ($\checkmark$?)

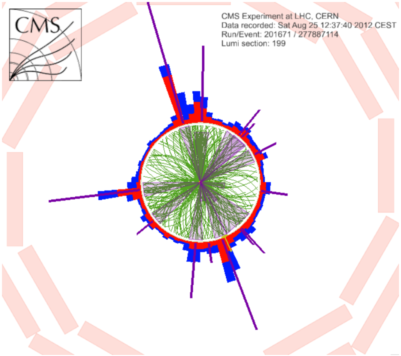
Motivation for Tree Level Matrix Elements (MEs)


CMS
CMS Experiment at LHC, CERN
Data recorded: Sat Aug 25 12:37:40 2012 CEST
Run/Event: 201671 / 277887114
Lumi section: 199

13 jet event with $p_{\mathrm{T,min}} = 50$ GeV at $\sqrt{s} = 8$ TeV of 12 fb^{-1} set (anti-k_{T}; $R = 0.5$)
[CMS–EXO–12–009]

- Need MEs for very high multiplicities to test the SM and its possible extensions
- MEs at tree level are part of every simulated event:
  - Correction of the shower to account for interferences (Merging)
  - Real emissions as part of the higher order computations: N^{k}LO implies up to k additional particles
- Automatized, efficient ME generators work in principle for every multiplicity: ALPHA [Caravaglios, Moretti 1995], HELAC [Kanaki and Papadopoulos 2000], O'MEGA [Moretti, Ohl, and Reuter 2001], COMIX [Gleisberg and Höche 2008], . . . ($\checkmark$?)

Motivation for Tree Level Matrix Elements (MEs)



13 jet event with $p_{\mathrm{T,min}} = 50\,\mathrm{GeV}$ at $\sqrt{s} = 8\,\mathrm{TeV}$ of $12\,\mathrm{fb}^{-1}$ set (anti-k_T; $R = 0.5$)

[CMS–EXO–12–009]

- Need MEs for very high multiplicities to test the SM and its possible extensions
- MEs at tree level are part of every simulated event:
  - Correction of the shower to account for interferences (Merging)
  - Real emissions as part of the higher order computations: N^{k}LO implies up to k additional particles
- Automatized, efficient ME generators work in principle for every multiplicity: ALPHA [Caravaglios, Moretti 1995], HELAC [Kanaki and Papadopoulos 2000], O'MEGA [Moretti, Ohl, and Reuter 2001], COMIX [Gleisberg and Höche 2008], ... ($\checkmark$?)

Motivation for a Virtual Machine (VM)

- Direct numerical implementations of recursion relations are usually less flexible.
  Only recently the first program of this type has been extended to more general theories [Höche, Kuttimalai, Schumann, and Siegert 2014]

- Popular (traditional) method to combine

$$\text{fast code} + \text{full flexibility}$$
$$= \text{meta programming}$$

  - Determine the minimal, algebraic expression in a high-level language like `Form`, `Mathematica`, `OCaml` or `Python`
  - Evaluate this in a numerical fast language like `C` or `Fortran`
  - Examples: MadGraph [Alwall et al. 2011], FormCalc [Hahn and Pérez-Victoria 1999], GoSam [Cullen et al. 2014], O'Mega ...

- ↯ However, analytic expressions of complicated multi-jet events can easily reach gigabyte sizes ↯

Motivation for a Virtual Machine (VM)

- Direct numerical implementations of recursion relations are usually less flexible.
  Only recently the first program of this type has been extended to more general theories [Höche, Kuttimalai, Schumann, and Siegert 2014]

- Popular (traditional) method to combine

$$\text{fast code} + \text{full flexibility}$$
$$= \text{meta programming}$$

  - Determine the minimal, algebraic expression in a high-level language like `Form`, `Mathematica`, `OCaml` or `Python`
  - Evaluate this in a numerical fast language like `C` or `Fortran`
  - Examples: MadGraph [Alwall et al. 2011],
    FormCalc [Hahn and Pérez-Victoria 1999], GoSam [Cullen et al. 2014],
    O'Mega ...

- ↯ However, analytic expressions of complicated multi-jet events can easily reach gigabyte sizes ↯

Motivation for a Virtual Machine (VM)

- Direct numerical implementations of recursion relations are usually less flexible.
  Only recently the first program of this type has been extended to more general theories [Höche, Kuttimalai, Schumann, and Siegert 2014]

- Popular (traditional) method to combine

  $$\text{fast code} + \text{full flexibility}$$
  $$= \text{meta programming}$$

  - Determine the minimal, algebraic expression in a high-level language like `Form`, `Mathematica`, `OCaml` or `Python`
  - Evaluate this in a numerical fast language like `C` or `Fortran`
  - Examples: MadGraph [Alwall et al. 2011],
    FormCalc [Hahn and Pérez-Victoria 1999], GoSam [Cullen et al. 2014],
    O'Mega ...

- ↯ However, analytic expressions of complicated multi-jet events can easily reach gigabyte sizes ↯

Motivation for a Virtual Machine (VM)

- $2g \to 6g$ process gives for all color-flows [Kilian, Ohl, Reuter, and Speckner 2012] $\sim 4\,\mathrm{GB}$ `Fortran` code.
  Code of this size either fails to compile and link or needs several days
- No promising path to ever higher multiplicities . . .
- Possible solution: With a virtual machine (VM) you circumvent the compilation of the large source code completely
- A VM is furthermore simple to implement and to parallelize and offers similar performance as compiled code [Chokoufe Nejad, Ohl, and Reuter 2014]
- A VM is in our context no `OS` emulation or similar stuff

Motivation for a Virtual Machine (VM)

- $2g \to 6g$ process gives for all color-flows [Kilian, Ohl, Reuter, and Speckner 2012] $\sim 4\,\mathrm{GB}$ `Fortran` code.
  Code of this size either fails to compile and link or needs several days
- No promising path to ever higher multiplicities . . .
- Possible solution: With a virtual machine (VM) you circumvent the compilation of the large source code completely
- A VM is furthermore simple to implement and to parallelize and offers similar performance as compiled code [Chokoufe Nejad, Ohl, and Reuter 2014]
- A VM is in our context no `OS` emulation or similar stuff

Outline

General Aspects

What is a VM?

- A VM is in our context compiled program (interpreter)
- It is able to read instructions from disk and perform an arbitrary number of operations of a finite instruction set
- Instructions can be saved as byte code encoded in mere numbers in a simple ASCII file
- Imagine the VM as a machine that is given registers and instructions what to do with them
- Just like a `CPU` but on a higher level as the registers are e.g. arrays of momenta and wave functions and the instructions scalar products

What to put in the Byte Code?

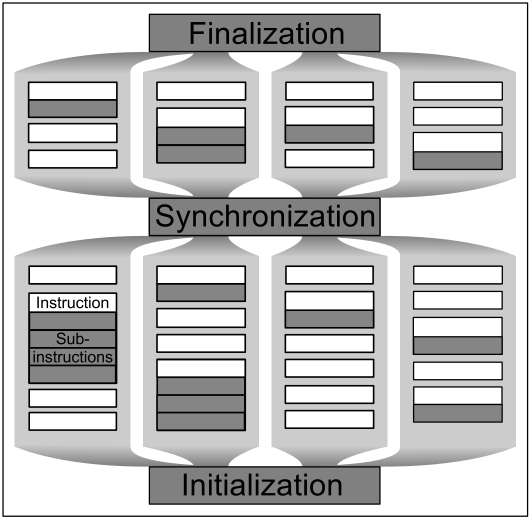
- To construct the VM dynamically, we have to supply a header with the numbers of objects that have to be allocated
- Optionally we can give version numbers to specify the physical model, comments how the byte code was created and tables with precomputed properties like information over spin, color and flavor
- This is followed by the body of instructions with the nontrivial information how to compute a process
- The first object of an instruction is the opcode that specifies which operation is executed. This is usually followed by adresses, e.g.

  `1 7 4 3` $\Leftrightarrow$ `momentum(7) = momentum(4) + momentum(3)`

Interpreter

- Simple program that reads at first the byte code into memory
- Loops over instructions with a decode function and given input values
- Translation of the byte code to machine code is fast compared to the execution (multiple complex valued scalar products)
- Adaption of the interpreter to a new kind of "problem" requires
  - Specification of static informations
  - Writing the `switch/case` statements of the decode function
- VM is quickly compiled once. Handy to check many small processes quickly and mandatory for very large ones

Organisation of the Parallelization in the VM



- Group instructions into building blocks to minimize the number of synchronization points
- Divide the computation into levels
- All building blocks commute in every level,

  i.e. only one thread is writing to a register per level

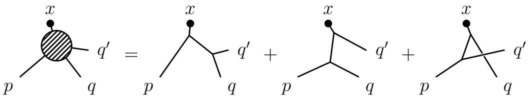
Parallelization in high energy physics

- We usually assume that we can trivially parallelize our computations by computing multiple phase space points at once (phase space = momenta, flavor, helizity, color)
- In extreme computations, the objects of a single phase space point might already fill up your cache
- Trying to compute multiple points at once can induce traffic jam between `RAM` and `CPU` and might be slower than single core performance
- The VM is a straightforward implementation of the parallel computation of a single phase space point

A Virtual Machine for O'Mega

Optimizing Matrix Element Generator

- O'MEGA [Moretti, Ohl, and Reuter 2001] computes amplitudes with 1 particle off shell wave functions W

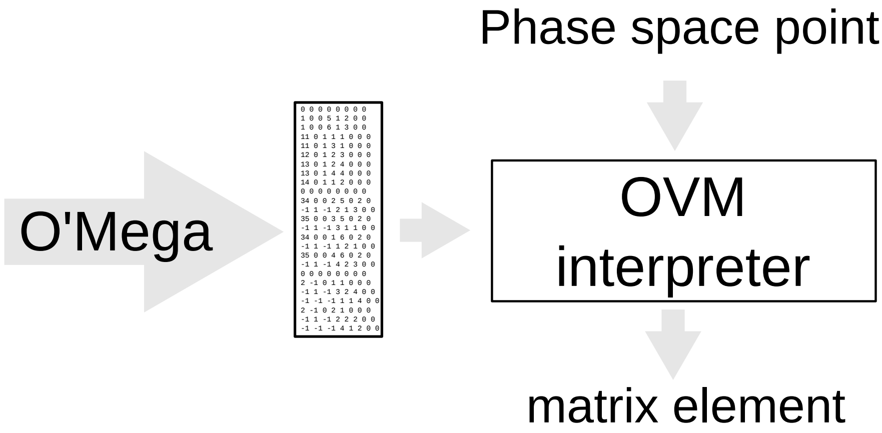


- On tree-level (and 1-loop level, [→ RECOLA talk by A. Denner]) you can construct the set of all currents recursively

- Finally you need some keystones K to replace the sum over Feynman diagrams, e.g.

$$\sum_{i=1}^{F(n)} D_i = \sum_{k,l,m=1}^{P(n)} K^3_{f_k f_l f_m}(p_k, p_l, p_m)\, W_{f_k}(p_k)\, W_{f_l}(p_l)\, W_{f_m}(p_m)$$

- The calculation froms a Directed Acyclical Graph (DAG), optimized by O'MEGA to obtain the minimal number of connections



```
0 0 0 0 0 0 0 0
1 0 0 5 1 2 0 0
1 0 0 6 1 3 0 0
11 0 1 1 1 0 0 0
11 0 1 3 1 0 0 0
12 0 1 2 3 0 0 0
13 0 1 2 4 0 0 0
13 0 1 4 4 0 0 0
14 0 1 1 2 0 0 0
0 0 0 0 0 0 0 0
34 0 0 2 5 0 2 0
-1 1 -1 2 1 3 0 0
35 0 0 3 5 0 2 0
-1 1 -1 3 1 1 0 0
34 0 0 1 6 0 2 0
-1 1 -1 1 2 1 0 0
35 0 0 4 6 0 2 0
-1 1 -1 4 2 3 0 0
0 0 0 0 0 0 0 0
2 -1 0 1 1 0 0 0
-1 1 -1 3 2 4 0 0
-1 -1 -1 1 1 4 0 0
2 -1 0 2 1 0 0 0
-1 1 -1 2 2 2 0 0
-1 -1 -1 4 1 2 0 0
```

Byte Code Creation in O'MEGA / OCaml

- Feynman rules form a finite number of instructions and are therefore good candidates for the translation into byte code
- OCaml can compare abstract objects like wave functions, momenta or amplitudes
- Fortran arrays identify their objects via their index
- Take the set of objects and apply a mapping to natural numbers using the given order

code	coupl	coeff	lhs	rhs_1	rhs_2	rhs_3	rhs_4
ADD_MOMENTA	0	0	p_lhs	p_rhs_1	p_rhs_2	p_rhs_3	0
LOAD_X	PDG	0	wf	outer _ind	0	0	amp
PROPAGATE_Y	PDG	width	wf	p	0	0	amp
FUSE_Z	coupl	coeff	lhs	rhs_1	rhs_2	rhs_3	rhs_4
CALC_BRAKET	sign	0	amp	sym	0	0	0

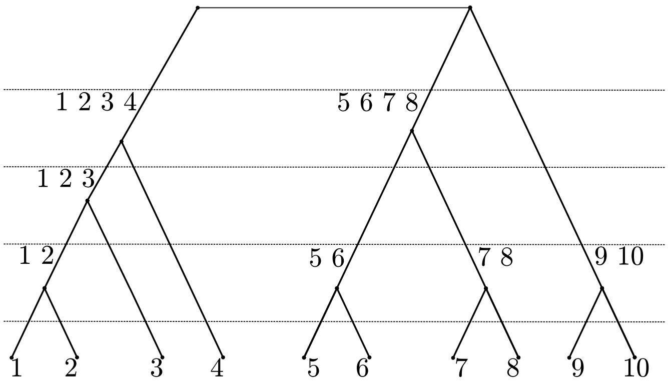
Performance of byte code generation

- Byte code can be faster produced using less `RAM` and is smaller than native `Fortran` source code
- For $gg \to 6g$
  memory requirements are reduced from 2.17 GB to 1.34 GB and the time to produce it from 11 min 52 s zu 3 min 35 s

process	BC size	`Fortran` size	t_{compile}
$gg \to gggggg$	428 MB	4.0 GB	-
$gg \to ggggg$	9.4 MB	85 MB	483(18) s
$gg \to q\bar{q}q'\bar{q}'q''\bar{q}''g$	3.2 MB	27 MB	166(15) s
$e^+e^- \to e^+e^-e^+e^-e^+e^-e^+e^-e^+e^-$	0.7 MB	1.9 MB	32.46(13) s

- For smaller processes no big changes

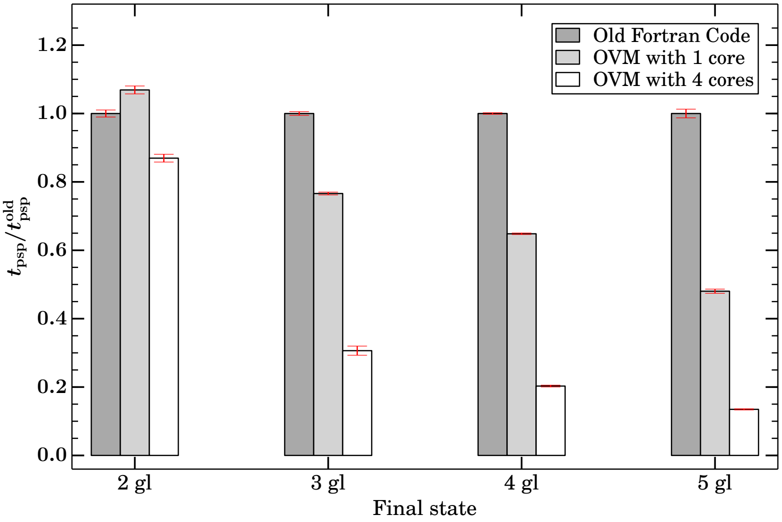
Parallelization in recursive computations



Identify a level by counting external momenta

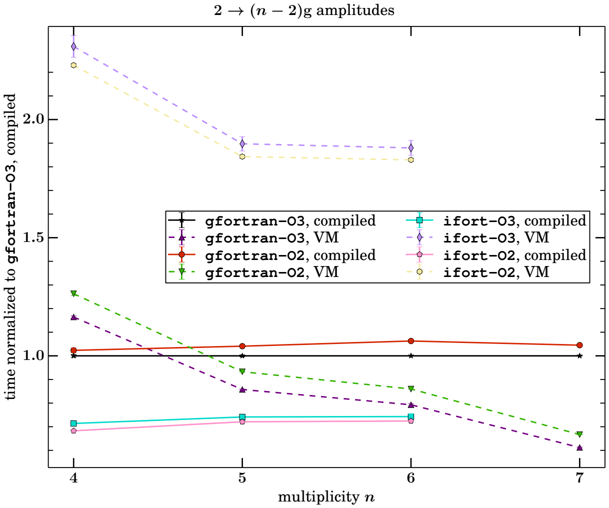
Example of one tree of a large set

Speed of matrix elements - First Look



[gfortran 4.7, Intel i7-2720QM @ 2.20GHz]

Speed of matrix elements - Now with two compilers



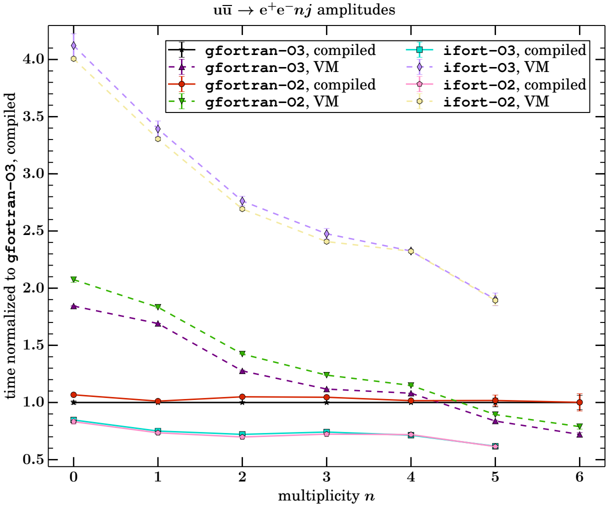
[gfortran 4.7, ifort 14, Intel Xeon E5-2440 @ 2.40GHz]

Both VMs improve with increasing multiplicity

`ifort` has large offset for the VM, could be solved with profile-guided optimization

`ifort` fails to compile the $2 \to 5$ gluon amplitude (even with -O0)

Speed of matrix elements - Explaining the Scaling



Same scaling behavior

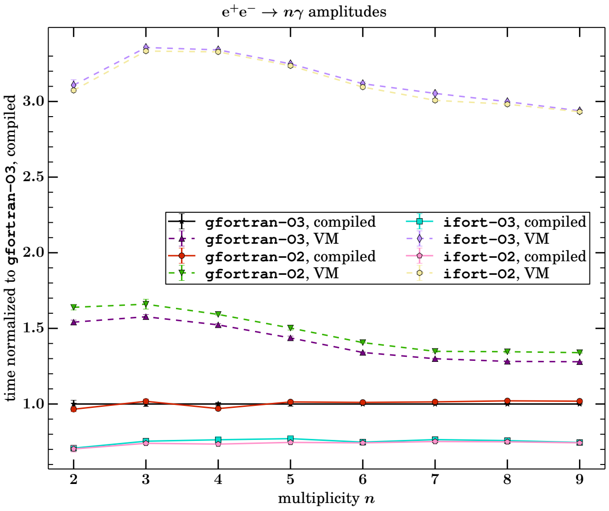
Virtualization costs are constant

VM does loop over levels and instructions therein

Source code is like a unrolled version of this loop

Double loop has higher probability to keep decode function in instruction cache

Speed of matrix elements



Improvements for VM are smaller with increasing multiplicity for pure QED

Less to be done on more wave functions per level

Unrolled version can gain more from data prefetching

Note: $e^+e^- \to 9\gamma$ gives 125 KiB, $gg \to 4g$ gives 269 KiB

Quantitative Analysis of the Parallelization

- Amdahl's Law divides an algorithm into parallelizable parts p and strictly serial parts $1 - p$

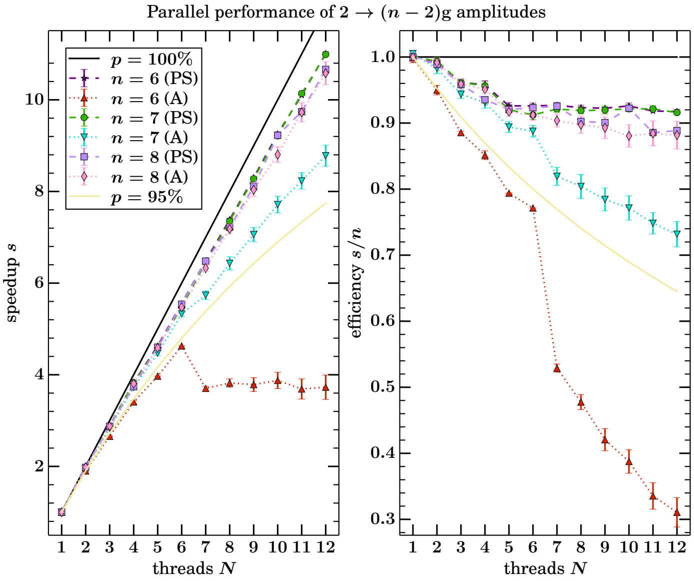
- The possible speedup s for a computation with n cores is then

$$s \equiv \frac{t^{(1)}}{t^{(n)}} = \frac{1}{(1-p) + \frac{p}{n}} \leq \frac{1}{1-p}$$

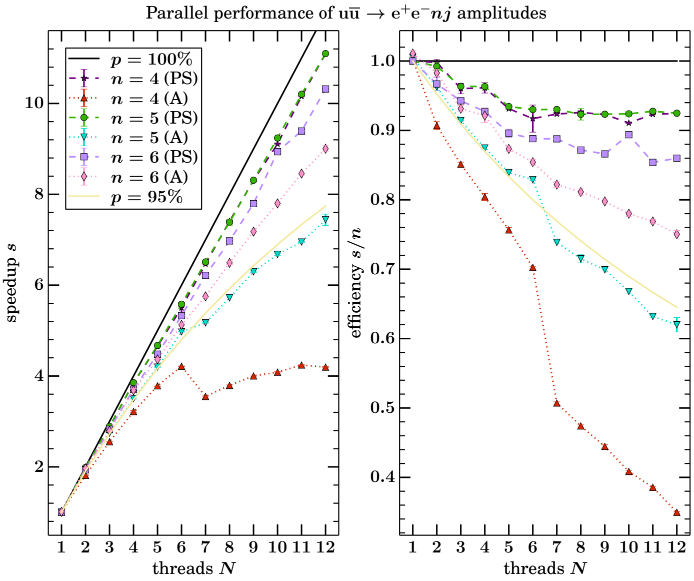
- Idealized version since communication costs between cores $\mathcal{O}\left(n\right)$ have been neglected in the denominator

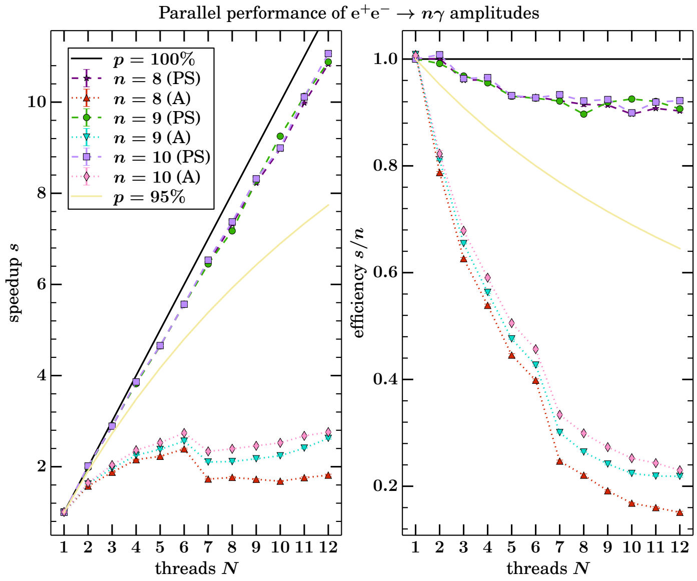
- Compare parallel evaluation of the amplitude (A) with the computation of multiple phase space points at once (PS)

Quantitative Analysis of the Parallelization



Quantitative Analysis of the Parallelization




Parallel performance of $e^+e^- \to n\gamma$ amplitudes
$p = 100\%$
$n = 8$ (PS)
$n = 8$ (A)
$n = 9$ (PS)
$n = 9$ (A)
$n = 10$ (PS)
$n = 10$ (A)
$p = 95\%$
speedup s
efficiency s/n
threads N
threads N

Paving the way for matrix elements on GPUs?

- A VM could be the perfect tool for computations on the `GPU`, as it avoids the finite kernel size problem
- Previous studies show degrading in performance with growing number of external particles and have to be split in smaller programs [Hagiwara et al. 2013]
- Decode function remains small for arbitrary processes. Efficiency of memory management may remain an obstacle
- You could transfer instructions and VM once to the GPU, to reduce communication costs
- For event generation, only send the outer quantum number to the `GPU` and receive the amplitude. Phase space integration can happen on the `CPU`

How to use it?

```
$method = ovm    # omega
```

Implemented since `Whizard 2.2.3` and directly usable for `2HDM_CKM, 2HDM, HSExt, QCD, QED, SM_CKM, SM_Higgs, SM, Zprime`

More models with general Lorentz structures or upon request

Summary

- A virtual machine allows to compute processes directly without the need to compile for hours/days
- Execution times can be faster or slower than compiled code but is always in the same order of magnitude
- Implementation is based on tree-level matrix elements but the idea is applicable in general
- Parallelization of single phase space points might be necessary for very complex processes and is straightforward with the VM

References I

Johan Alwall et al. MadGraph 5: going beyond. In *Journal of High Energy Physics*, **2011**:6 (June 2011), p. 128. arXiv: 1106.0522v1

Bijan Chokoufe Nejad, Thorsten Ohl, and Juergen Reuter. Simple, Parallel, High-Performance Virtual Machines for Extreme Computations. In (2014). arXiv: 1411.3834

Gavin Cullen et al. *GoSam-2.0: a tool for automated one-loop calculations within the Standard Model and beyond*. Apr. 2014. arXiv: 1404.7096

Tanju Gleisberg and Stefan Höche. Comix, a new matrix element generator. In *Journal of High Energy Physics*, **2008**:12 (Dec. 2008), pp. 039–039. arXiv: 0808.3674v2

K Hagiwara et al. Fast computation of MadGraph amplitudes on graphics processing unit (GPU). In (May 2013), p. 37. arXiv: 1305.0708

T. Hahn and M. Pérez-Victoria. Automated one-loop calculations in four and D dimensions. In *Computer Physics Communications*, **118**:2-3 (May 1999), pp. 153–165. arXiv: hep-ph/9807565 [hep-ph]

Stefan Höche, Silvan Kuttimalai, Steffen Schumann, and Frank Siegert. Beyond Standard Model calculations with Sherpa. In *arXiv*, 1412.6478 (Dec. 2014). arXiv: 1412.6478

References II

Aggeliki Kanaki and Costas G. Papadopoulos. HELAC: A package to compute electroweak helicity amplitudes. In*Computer Physics Communications*, **132**:3 (Nov. 2000), pp. 306–315. arXiv: 0002082 [hep-ph]

W. Kilian, T. Ohl, J. Reuter, and C. Speckner. QCD in the color-flow representation. In*Journal of High Energy Physics*, **2012**:10 (Oct. 2012), p. 22. arXiv: 1206.3700v2

Mauro Moretti, Thorsten Ohl, and Juergen Reuter. O'Mega: An Optimizing Matrix Element Generator. In*arXiv*, **hep-ph**:0102195 (Feb. 2001). arXiv: hep-ph/0102195 [hep-ph]

Backup

Implementation of parallelization

```
subroutine iterate_instructions (vm)
  type(vm_t), intent(inout) :: vm
  integer :: instruction, level
  !$omp parallel
  do level = 1, vm%N_levels - 1
     !$omp do schedule (static)
     do instruction = vm%levels (level) + 1, vm%levels (level + 1)
        call decode (vm, instruction)
     end do
     !$omp end do
  end do
  !$omp end parallel
end subroutine iterate_instructions
```

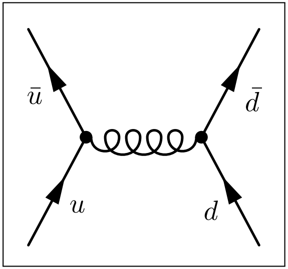
But also the color sum had to be parallelized via

```
!$omp parallel do reduction(+:amp2)
```

Byte code in detail

```
Bytecode file generated automatically by O'Mega for OVM.
Do not delete any lines. You called O'Mega with
  /home/bijan/Dropbox/MasterThesis/Build/bin/omega_QCD.opt -scatter "u ubar -> d dbar"

N_mom N_prt N_in N_out N_amp N_coupl N_hel N_cflow N_cind N_cfactors
5 4 2 2 2 3 16 2 2 4
N_flv N_psi N_psibar N_vec
1 4 2 2 0 0 0 0 0 0
Spin states table
-1 -1 -1 -1
-1 -1 -1 1
-1 -1 1 -1
-1 -1 1 1
-1 1 -1 -1
-1 1 -1 1
-1 1 1 -1
-1 1 1 1
1 -1 -1 -1
1 -1 -1 1
1 -1 1 -1
1 -1 1 1
1 1 -1 -1
1 1 -1 1
1 1 1 -1
1 1 1 1
Color flows table: [ (i, j) (k, l) -> (m, n) ...]
1 0 0 -1 2 0 0 -2
2 0 0 -1 2 0 0 -1
Color factors table: [ i, j: num den power], where i, j are the indexed color flows.
1 1 1 1 2
1 2 1 1 1
2 1 1 1 1
2 2 1 1 2
```



$$\text{cf} = \frac{\text{num}}{\text{den}} N^{\text{pwr}}$$

Byte code in detail

```
OVM instructions
0 0 0 0 0 0 0 0
1 0 0 5 1 2 0 0
11 2 0 1 1 0 0 1
14 -2 0 1 2 0 0 1
12 -1 0 2 3 0 0 1
13 1 0 4 4 0 0 1
11 2 0 2 1 0 0 2
14 -2 0 1 2 0 0 2
12 -1 0 2 3 0 0 2
13 1 0 3 4 0 0 2
0 0 0 0 0 0 0 0
34 21 2 1 5 0 0 2
-1 1 -1 1 1 2 0 0
35 21 2 2 5 0 0 1
-1 1 -1 2 1 1 0 0
0 0 0 0 0 0 0 0
2 -1 0 1 1 0 0 0
-1 1 -1 2 2 4 0 0
2 -1 0 2 1 0 0 0
-1 1 -1 1 2 3 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

```fortran
integer, parameter :: ovm_LOAD_U = 11
integer, parameter :: ovm_LOAD_UBAR = 12
integer, parameter :: ovm_LOAD_V = 13
integer, parameter :: ovm_LOAD_VBAR = 14
integer, parameter :: ovm_LOAD_VECTOR = 15
integer, parameter :: ovm_LOAD_CONJ_VECTOR = 16

integer, parameter :: ovm_ADD_MOMENTA = 1
integer, parameter :: ovm_CALC_BRAKET = 2

integer, parameter :: ovm_PROPAGATE_PSI = 31
integer, parameter :: ovm_PROPAGATE_PSIBAR = 32
integer, parameter :: ovm_PROPAGATE_UNITARITY = 33
integer, parameter :: ovm_PROPAGATE_FEYNMAN = 34
integer, parameter :: ovm_PROPAGATE_COL_FEYNMAN = 35

integer, parameter :: ovm_FUSE_VEC_PSIBAR_PSI = -1
integer, parameter :: ovm_FUSE_PSI_VEC_PSI = -2
integer, parameter :: ovm_FUSE_PSIBAR_PSIBAR_VEC = -3
integer, parameter :: ovm_FUSE_GLU_GLU_GLU = -4
integer, parameter :: ovm_FUSE_WFS_V4 = -5
```