Integrated Matrix Extension ISA

Proposal G

Erich Focht (Openchip)

Building on «Proposal A» with Marc Casas (BSC)

With contributions from Philipp Tomsich (VRULL)

24/02/2025

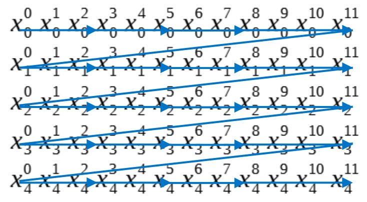
MATRIX IN MEMORY

$$
\begin{array}{cccccccccccccccccccccccccccccccc}
x_{0}^{0} & x_{0}^{1} & x_{0}^{2} & x_{0}^{3} & x_{0}^{4} & x_{0}^{5} & x_{0}^{6} & x_{0}^{7} & x_{0}^{8} & x_{0}^{9} & x_{0}^{10} & x_{0}^{11} & x_{0}^{12} & x_{0}^{13} & x_{0}^{14} & x_{0}^{15} & x_{0}^{16} & x_{0}^{17} & x_{0}^{18} & x_{0}^{19} & x_{0}^{20} & x_{0}^{21} & x_{0}^{22} & x_{0}^{23} & x_{0}^{24} & x_{0}^{25} & x_{0}^{26} & x_{0}^{27} & x_{0}^{28} & x_{0}^{29} & x_{0}^{30} & x_{0}^{31} \\
x_{1}^{0} & x_{1}^{1} & x_{1}^{2} & x_{1}^{3} & x_{1}^{4} & x_{1}^{5} & x_{1}^{6} & x_{1}^{7} & x_{1}^{8} & x_{1}^{9} & x_{1}^{10} & x_{1}^{11} & x_{1}^{12} & x_{1}^{13} & x_{1}^{14} & x_{1}^{15} & x_{1}^{16} & x_{1}^{17} & x_{1}^{18} & x_{1}^{19} & x_{1}^{20} & x_{1}^{21} & x_{1}^{22} & x_{1}^{23} & x_{1}^{24} & x_{1}^{25} & x_{1}^{26} & x_{1}^{27} & x_{1}^{28} & x_{1}^{29} & x_{1}^{30} & x_{1}^{31} \\
x_{2}^{0} & x_{2}^{1} & x_{2}^{2} & x_{2}^{3} & x_{2}^{4} & x_{2}^{5} & x_{2}^{6} & x_{2}^{7} & x_{2}^{8} & x_{2}^{9} & x_{2}^{10} & x_{2}^{11} & x_{2}^{12} & x_{2}^{13} & x_{2}^{14} & x_{2}^{15} & x_{2}^{16} & x_{2}^{17} & x_{2}^{18} & x_{2}^{19} & x_{2}^{20} & x_{2}^{21} & x_{2}^{22} & x_{2}^{23} & x_{2}^{24} & x_{2}^{25} & x_{2}^{26} & x_{2}^{27} & x_{2}^{28} & x_{2}^{29} & x_{2}^{30} & x_{2}^{31} \\
x_{3}^{0} & x_{3}^{1} & x_{3}^{2} & x_{3}^{3} & x_{3}^{4} & x_{3}^{5} & x_{3}^{6} & x_{3}^{7} & x_{3}^{8} & x_{3}^{9} & x_{3}^{10} & x_{3}^{11} & x_{3}^{12} & x_{3}^{13} & x_{3}^{14} & x_{3}^{15} & x_{3}^{16} & x_{3}^{17} & x_{3}^{18} & x_{3}^{19} & x_{3}^{20} & x_{3}^{21} & x_{3}^{22} & x_{3}^{23} & x_{3}^{24} & x_{3}^{25} & x_{3}^{26} & x_{3}^{27} & x_{3}^{28} & x_{3}^{29} & x_{3}^{30} & x_{3}^{31} \\
x_{4}^{0} & x_{4}^{1} & x_{4}^{2} & x_{4}^{3} & x_{4}^{4} & x_{4}^{5} & x_{4}^{6} & x_{4}^{7} & x_{4}^{8} & x_{4}^{9} & x_{4}^{10} & x_{4}^{11} & x_{4}^{12} & x_{4}^{13} & x_{4}^{14} & x_{4}^{15} & x_{4}^{16} & x_{4}^{17} & x_{4}^{18} & x_{4}^{19} & x_{4}^{20} & x_{4}^{21} & x_{4}^{22} & x_{4}^{23} & x_{4}^{24} & x_{4}^{25} & x_{4}^{26} & x_{4}^{27} & x_{4}^{28} & x_{4}^{29} & x_{4}^{30} & x_{4}^{31} \\
x_{5}^{0} & x_{5}^{1} & x_{5}^{2} & x_{5}^{3} & x_{5}^{4} & x_{5}^{5} & x_{5}^{6} & x_{5}^{7} & x_{5}^{8} & x_{5}^{9} & x_{5}^{10} & x_{5}^{11} & x_{5}^{12} & x_{5}^{13} & x_{5}^{14} & x_{5}^{15} & x_{5}^{16} & x_{5}^{17} & x_{5}^{18} & x_{5}^{19} & x_{5}^{20} & x_{5}^{21} & x_{5}^{22} & x_{5}^{23} & x_{5}^{24} & x_{5}^{25} & x_{5}^{26} & x_{5}^{27} & x_{5}^{28} & x_{5}^{29} & x_{5}^{30} & x_{5}^{31} \\
x_{6}^{0} & x_{6}^{1} & x_{6}^{2} & x_{6}^{3} & x_{6}^{4} & x_{6}^{5} & x_{6}^{6} & x_{6}^{7} & x_{6}^{8} & x_{6}^{9} & x_{6}^{10} & x_{6}^{11} & x_{6}^{12} & x_{6}^{13} & x_{6}^{14} & x_{6}^{15} & x_{6}^{16} & x_{6}^{17} & x_{6}^{18} & x_{6}^{19} & x_{6}^{20} & x_{6}^{21} & x_{6}^{22} & x_{6}^{23} & x_{6}^{24} & x_{6}^{25} & x_{6}^{26} & x_{6}^{27} & x_{6}^{28} & x_{6}^{29} & x_{6}^{30} & x_{6}^{31} \\
x_{7}^{0} & x_{7}^{1} & x_{7}^{2} & x_{7}^{3} & x_{7}^{4} & x_{7}^{5} & x_{7}^{6} & x_{7}^{7} & x_{7}^{8} & x_{7}^{9} & x_{7}^{10} & x_{7}^{11} & x_{7}^{12} & x_{7}^{13} & x_{7}^{14} & x_{7}^{15} & x_{7}^{16} & x_{7}^{17} & x_{7}^{18} & x_{7}^{19} & x_{7}^{20} & x_{7}^{21} & x_{7}^{22} & x_{7}^{23} & x_{7}^{24} & x_{7}^{25} & x_{7}^{26} & x_{7}^{27} & x_{7}^{28} & x_{7}^{29} & x_{7}^{30} & x_{7}^{31} \\
x_{8}^{0} & x_{8}^{1} & x_{8}^{2} & x_{8}^{3} & x_{8}^{4} & x_{8}^{5} & x_{8}^{6} & x_{8}^{7} & x_{8}^{8} & x_{8}^{9} & x_{8}^{10} & x_{8}^{11} & x_{8}^{12} & x_{8}^{13} & x_{8}^{14} & x_{8}^{15} & x_{8}^{16} & x_{8}^{17} & x_{8}^{18} & x_{8}^{19} & x_{8}^{20} & x_{8}^{21} & x_{8}^{22} & x_{8}^{23} & x_{8}^{24} & x_{8}^{25} & x_{8}^{26} & x_{8}^{27} & x_{8}^{28} & x_{8}^{29} & x_{8}^{30} & x_{8}^{31} \\
x_{9}^{0} & x_{9}^{1} & x_{9}^{2} & x_{9}^{3} & x_{9}^{4} & x_{9}^{5} & x_{9}^{6} & x_{9}^{7} & x_{9}^{8} & x_{9}^{9} & x_{9}^{10} & x_{9}^{11} & x_{9}^{12} & x_{9}^{13} & x_{9}^{14} & x_{9}^{15} & x_{9}^{16} & x_{9}^{17} & x_{9}^{18} & x_{9}^{19} & x_{9}^{20} & x_{9}^{21} & x_{9}^{22} & x_{9}^{23} & x_{9}^{24} & x_{9}^{25} & x_{9}^{26} & x_{9}^{27} & x_{9}^{28} & x_{9}^{29} & x_{9}^{30} & x_{9}^{31} \\
x_{10}^{0} & x_{10}^{1} & x_{10}^{2} & x_{10}^{3} & x_{10}^{4} & x_{10}^{5} & x_{10}^{6} & x_{10}^{7} & x_{10}^{8} & x_{10}^{9} & x_{10}^{10} & x_{10}^{11} & x_{10}^{12} & x_{10}^{13} & x_{10}^{14} & x_{10}^{15} & x_{10}^{16} & x_{10}^{17} & x_{10}^{18} & x_{10}^{19} & x_{10}^{20} & x_{10}^{21} & x_{10}^{22} & x_{10}^{23} & x_{10}^{24} & x_{10}^{25} & x_{10}^{26} & x_{10}^{27} & x_{10}^{28} & x_{10}^{29} & x_{10}^{30} & x_{10}^{31} \\
x_{11}^{0} & x_{11}^{1} & x_{11}^{2} & x_{11}^{3} & x_{11}^{4} & x_{11}^{5} & x_{11}^{6} & x_{11}^{7} & x_{11}^{8} & x_{11}^{9} & x_{11}^{10} & x_{11}^{11} & x_{11}^{12} & x_{11}^{13} & x_{11}^{14} & x_{11}^{15} & x_{11}^{16} & x_{11}^{17} & x_{11}^{18} & x_{11}^{19} & x_{11}^{20} & x_{11}^{21} & x_{11}^{22} & x_{11}^{23} & x_{11}^{24} & x_{11}^{25} & x_{11}^{26} & x_{11}^{27} & x_{11}^{28} & x_{11}^{29} & x_{11}^{30} & x_{11}^{31} \\
x_{12}^{0} & x_{12}^{1} & x_{12}^{2} & x_{12}^{3} & x_{12}^{4} & x_{12}^{5} & x_{12}^{6} & x_{12}^{7} & x_{12}^{8} & x_{12}^{9} & x_{12}^{10} & x_{12}^{11} & x_{12}^{12} & x_{12}^{13} & x_{12}^{14} & x_{12}^{15} & x_{12}^{16} & x_{12}^{17} & x_{12}^{18} & x_{12}^{19} & x_{12}^{20} & x_{12}^{21} & x_{12}^{22} & x_{12}^{23} & x_{12}^{24} & x_{12}^{25} & x_{12}^{26} & x_{12}^{27} & x_{12}^{28} & x_{12}^{29} & x_{12}^{30} & x_{12}^{31} \\
x_{13}^{0} & x_{13}^{1} & x_{13}^{2} & x_{13}^{3} & x_{13}^{4} & x_{13}^{5} & x_{13}^{6} & x_{13}^{7} & x_{13}^{8} & x_{13}^{9} & x_{13}^{10} & x_{13}^{11} & x_{13}^{12} & x_{13}^{13} & x_{13}^{14} & x_{13}^{15} & x_{13}^{16} & x_{13}^{17} & x_{13}^{18} & x_{13}^{19} & x_{13}^{20} & x_{13}^{21} & x_{13}^{22} & x_{13}^{23} & x_{13}^{24} & x_{13}^{25} & x_{13}^{26} & x_{13}^{27} & x_{13}^{28} & x_{13}^{29} & x_{13}^{30} & x_{13}^{31} \\
x_{14}^{0} & x_{14}^{1} & x_{14}^{2} & x_{14}^{3} & x_{14}^{4} & x_{14}^{5} & x_{14}^{6} & x_{14}^{7} & x_{14}^{8} & x_{14}^{9} & x_{14}^{10} & x_{14}^{11} & x_{14}^{12} & x_{14}^{13} & x_{14}^{14} & x_{14}^{15} & x_{14}^{16} & x_{14}^{17} & x_{14}^{18} & x_{14}^{19} & x_{14}^{20} & x_{14}^{21} & x_{14}^{22} & x_{14}^{23} & x_{14}^{24} & x_{14}^{25} & x_{14}^{26} & x_{14}^{27} & x_{14}^{28} & x_{14}^{29} & x_{14}^{30} & x_{14}^{31} \\
x_{15}^{0} & x_{15}^{1} & x_{15}^{2} & x_{15}^{3} & x_{15}^{4} & x_{15}^{5} & x_{15}^{6} & x_{15}^{7} & x_{15}^{8} & x_{15}^{9} & x_{15}^{10} & x_{15}^{11} & x_{15}^{12} & x_{15}^{13} & x_{15}^{14} & x_{15}^{15} & x_{15}^{16} & x_{15}^{17} & x_{15}^{18} & x_{15}^{19} & x_{15}^{20} & x_{15}^{21} & x_{15}^{22} & x_{15}^{23} & x_{15}^{24} & x_{15}^{25} & x_{15}^{26} & x_{15}^{27} & x_{15}^{28} & x_{15}^{29} & x_{15}^{30} & x_{15}^{31}
\end{array}
$$

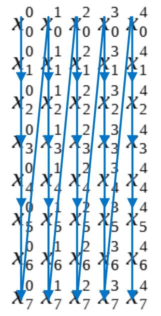
MATRIX IN MEMORY

$$\begin{matrix}
x_0^{0} & x_0^{1} & x_0^{2} & x_0^{3} & x_0^{4} & x_0^{5} & x_0^{6} & x_0^{7} & x_0^{8} & x_0^{9} & x_0^{10} & x_0^{11} & x_0^{12} & x_0^{13} & x_0^{14} & x_0^{15} & x_0^{16} & x_0^{17} & x_0^{18} & x_0^{19} & x_0^{20} & x_0^{21} & x_0^{22} & x_0^{23} & x_0^{24} & x_0^{25} & x_0^{26} & x_0^{27} & x_0^{28} & x_0^{29} & x_0^{30} & x_0^{31} \\
x_1^{0} & x_1^{1} & x_1^{2} & x_1^{3} & x_1^{4} & x_1^{5} & x_1^{6} & x_1^{7} & x_1^{8} & x_1^{9} & x_1^{10} & x_1^{11} & x_1^{12} & x_1^{13} & x_1^{14} & x_1^{15} & x_1^{16} & x_1^{17} & x_1^{18} & x_1^{19} & x_1^{20} & x_1^{21} & x_1^{22} & x_1^{23} & x_1^{24} & x_1^{25} & x_1^{26} & x_1^{27} & x_1^{28} & x_1^{29} & x_1^{30} & x_1^{31} \\
x_2^{0} & x_2^{1} & x_2^{2} & x_2^{3} & x_2^{4} & x_2^{5} & x_2^{6} & x_2^{7} & x_2^{8} & x_2^{9} & x_2^{10} & x_2^{11} & x_2^{12} & x_2^{13} & x_2^{14} & x_2^{15} & x_2^{16} & x_2^{17} & x_2^{18} & x_2^{19} & x_2^{20} & x_2^{21} & x_2^{22} & x_2^{23} & x_2^{24} & x_2^{25} & x_2^{26} & x_2^{27} & x_2^{28} & x_2^{29} & x_2^{30} & x_2^{31} \\
x_3^{0} & x_3^{1} & x_3^{2} & x_3^{3} & x_3^{4} & x_3^{5} & x_3^{6} & x_3^{7} & x_3^{8} & x_3^{9} & x_3^{10} & x_3^{11} & x_3^{12} & x_3^{13} & x_3^{14} & x_3^{15} & x_3^{16} & x_3^{17} & x_3^{18} & x_3^{19} & x_3^{20} & x_3^{21} & x_3^{22} & x_3^{23} & x_3^{24} & x_3^{25} & x_3^{26} & x_3^{27} & x_3^{28} & x_3^{29} & x_3^{30} & x_3^{31} \\
x_4^{0} & x_4^{1} & x_4^{2} & x_4^{3} & x_4^{4} & x_4^{5} & x_4^{6} & x_4^{7} & x_4^{8} & x_4^{9} & x_4^{10} & x_4^{11} & x_4^{12} & x_4^{13} & x_4^{14} & x_4^{15} & x_4^{16} & x_4^{17} & x_4^{18} & x_4^{19} & x_4^{20} & x_4^{21} & x_4^{22} & x_4^{23} & x_4^{24} & x_4^{25} & x_4^{26} & x_4^{27} & x_4^{28} & x_4^{29} & x_4^{30} & x_4^{31} \\
x_5^{0} & x_5^{1} & x_5^{2} & x_5^{3} & x_5^{4} & x_5^{5} & x_5^{6} & x_5^{7} & x_5^{8} & x_5^{9} & x_5^{10} & x_5^{11} & x_5^{12} & x_5^{13} & x_5^{14} & x_5^{15} & x_5^{16} & x_5^{17} & x_5^{18} & x_5^{19} & x_5^{20} & x_5^{21} & x_5^{22} & x_5^{23} & x_5^{24} & x_5^{25} & x_5^{26} & x_5^{27} & x_5^{28} & x_5^{29} & x_5^{30} & x_5^{31} \\
x_6^{0} & x_6^{1} & x_6^{2} & x_6^{3} & x_6^{4} & x_6^{5} & x_6^{6} & x_6^{7} & x_6^{8} & x_6^{9} & x_6^{10} & x_6^{11} & x_6^{12} & x_6^{13} & x_6^{14} & x_6^{15} & x_6^{16} & x_6^{17} & x_6^{18} & x_6^{19} & x_6^{20} & x_6^{21} & x_6^{22} & x_6^{23} & x_6^{24} & x_6^{25} & x_6^{26} & x_6^{27} & x_6^{28} & x_6^{29} & x_6^{30} & x_6^{31} \\
x_7^{0} & x_7^{1} & x_7^{2} & x_7^{3} & x_7^{4} & x_7^{5} & x_7^{6} & x_7^{7} & x_7^{8} & x_7^{9} & x_7^{10} & x_7^{11} & x_7^{12} & x_7^{13} & x_7^{14} & x_7^{15} & x_7^{16} & x_7^{17} & x_7^{18} & x_7^{19} & x_7^{20} & x_7^{21} & x_7^{22} & x_7^{23} & x_7^{24} & x_7^{25} & x_7^{26} & x_7^{27} & x_7^{28} & x_7^{29} & x_7^{30} & x_7^{31} \\
x_8^{0} & x_8^{1} & x_8^{2} & x_8^{3} & x_8^{4} & x_8^{5} & x_8^{6} & x_8^{7} & x_8^{8} & x_8^{9} & x_8^{10} & x_8^{11} & x_8^{12} & x_8^{13} & x_8^{14} & x_8^{15} & x_8^{16} & x_8^{17} & x_8^{18} & x_8^{19} & x_8^{20} & x_8^{21} & x_8^{22} & x_8^{23} & x_8^{24} & x_8^{25} & x_8^{26} & x_8^{27} & x_8^{28} & x_8^{29} & x_8^{30} & x_8^{31} \\
x_9^{0} & x_9^{1} & x_9^{2} & x_9^{3} & x_9^{4} & x_9^{5} & x_9^{6} & x_9^{7} & x_9^{8} & x_9^{9} & x_9^{10} & x_9^{11} & x_9^{12} & x_9^{13} & x_9^{14} & x_9^{15} & x_9^{16} & x_9^{17} & x_9^{18} & x_9^{19} & x_9^{20} & x_9^{21} & x_9^{22} & x_9^{23} & x_9^{24} & x_9^{25} & x_9^{26} & x_9^{27} & x_9^{28} & x_9^{29} & x_9^{30} & x_9^{31} \\
x_{10}^{0} & x_{10}^{1} & x_{10}^{2} & x_{10}^{3} & x_{10}^{4} & x_{10}^{5} & x_{10}^{6} & x_{10}^{7} & x_{10}^{8} & x_{10}^{9} & x_{10}^{10} & x_{10}^{11} & x_{10}^{12} & x_{10}^{13} & x_{10}^{14} & x_{10}^{15} & x_{10}^{16} & x_{10}^{17} & x_{10}^{18} & x_{10}^{19} & x_{10}^{20} & x_{10}^{21} & x_{10}^{22} & x_{10}^{23} & x_{10}^{24} & x_{10}^{25} & x_{10}^{26} & x_{10}^{27} & x_{10}^{28} & x_{10}^{29} & x_{10}^{30} & x_{10}^{31} \\
x_{11}^{0} & x_{11}^{1} & x_{11}^{2} & x_{11}^{3} & x_{11}^{4} & x_{11}^{5} & x_{11}^{6} & x_{11}^{7} & x_{11}^{8} & x_{11}^{9} & x_{11}^{10} & x_{11}^{11} & x_{11}^{12} & x_{11}^{13} & x_{11}^{14} & x_{11}^{15} & x_{11}^{16} & x_{11}^{17} & x_{11}^{18} & x_{11}^{19} & x_{11}^{20} & x_{11}^{21} & x_{11}^{22} & x_{11}^{23} & x_{11}^{24} & x_{11}^{25} & x_{11}^{26} & x_{11}^{27} & x_{11}^{28} & x_{11}^{29} & x_{11}^{30} & x_{11}^{31} \\
x_{12}^{0} & x_{12}^{1} & x_{12}^{2} & x_{12}^{3} & x_{12}^{4} & x_{12}^{5} & x_{12}^{6} & x_{12}^{7} & x_{12}^{8} & x_{12}^{9} & x_{12}^{10} & x_{12}^{11} & x_{12}^{12} & x_{12}^{13} & x_{12}^{14} & x_{12}^{15} & x_{12}^{16} & x_{12}^{17} & x_{12}^{18} & x_{12}^{19} & x_{12}^{20} & x_{12}^{21} & x_{12}^{22} & x_{12}^{23} & x_{12}^{24} & x_{12}^{25} & x_{12}^{26} & x_{12}^{27} & x_{12}^{28} & x_{12}^{29} & x_{12}^{30} & x_{12}^{31} \\
x_{13}^{0} & x_{13}^{1} & x_{13}^{2} & x_{13}^{3} & x_{13}^{4} & x_{13}^{5} & x_{13}^{6} & x_{13}^{7} & x_{13}^{8} & x_{13}^{9} & x_{13}^{10} & x_{13}^{11} & x_{13}^{12} & x_{13}^{13} & x_{13}^{14} & x_{13}^{15} & x_{13}^{16} & x_{13}^{17} & x_{13}^{18} & x_{13}^{19} & x_{13}^{20} & x_{13}^{21} & x_{13}^{22} & x_{13}^{23} & x_{13}^{24} & x_{13}^{25} & x_{13}^{26} & x_{13}^{27} & x_{13}^{28} & x_{13}^{29} & x_{13}^{30} & x_{13}^{31} \\
x_{14}^{0} & x_{14}^{1} & x_{14}^{2} & x_{14}^{3} & x_{14}^{4} & x_{14}^{5} & x_{14}^{6} & x_{14}^{7} & x_{14}^{8} & x_{14}^{9} & x_{14}^{10} & x_{14}^{11} & x_{14}^{12} & x_{14}^{13} & x_{14}^{14} & x_{14}^{15} & x_{14}^{16} & x_{14}^{17} & x_{14}^{18} & x_{14}^{19} & x_{14}^{20} & x_{14}^{21} & x_{14}^{22} & x_{14}^{23} & x_{14}^{24} & x_{14}^{25} & x_{14}^{26} & x_{14}^{27} & x_{14}^{28} & x_{14}^{29} & x_{14}^{30} & x_{14}^{31} \\
x_{15}^{0} & x_{15}^{1} & x_{15}^{2} & x_{15}^{3} & x_{15}^{4} & x_{15}^{5} & x_{15}^{6} & x_{15}^{7} & x_{15}^{8} & x_{15}^{9} & x_{15}^{10} & x_{15}^{11} & x_{15}^{12} & x_{15}^{13} & x_{15}^{14} & x_{15}^{15} & x_{15}^{16} & x_{15}^{17} & x_{15}^{18} & x_{15}^{19} & x_{15}^{20} & x_{15}^{21} & x_{15}^{22} & x_{15}^{23} & x_{15}^{24} & x_{15}^{25} & x_{15}^{26} & x_{15}^{27} & x_{15}^{28} & x_{15}^{29} & x_{15}^{30} & x_{15}^{31}
\end{matrix}$$

stride 1 access direction



row-major order

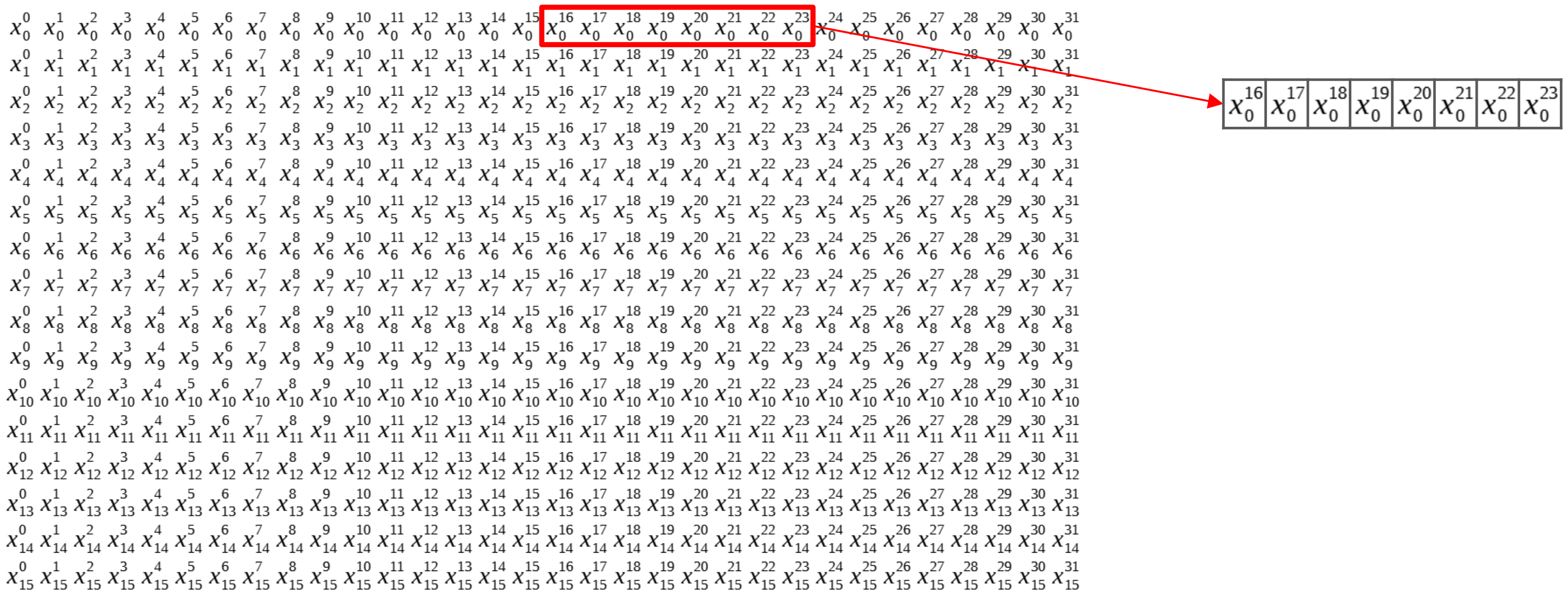


column-major order

MEMORY TO VREG TILE CONTENT

Matrix in memory – FP32

VLEN=256



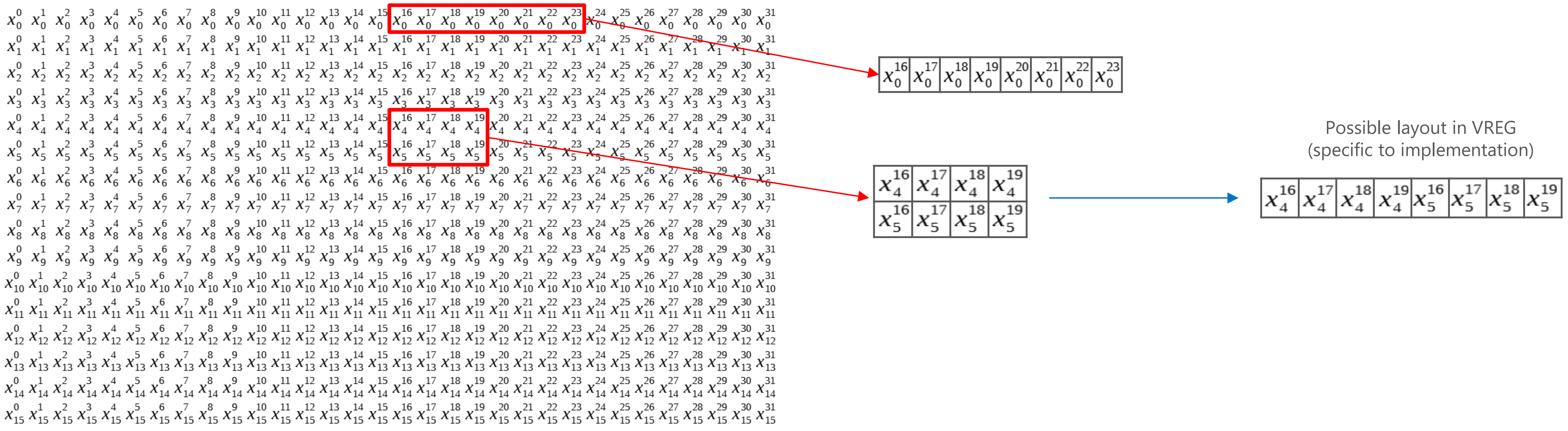
x_0^{16} x_0^{17} x_0^{18} x_0^{19} x_0^{20} x_0^{21} x_0^{22} x_0^{23}

MEMORY TO VREG TILE CONTENT

Matrix in memory – FP32

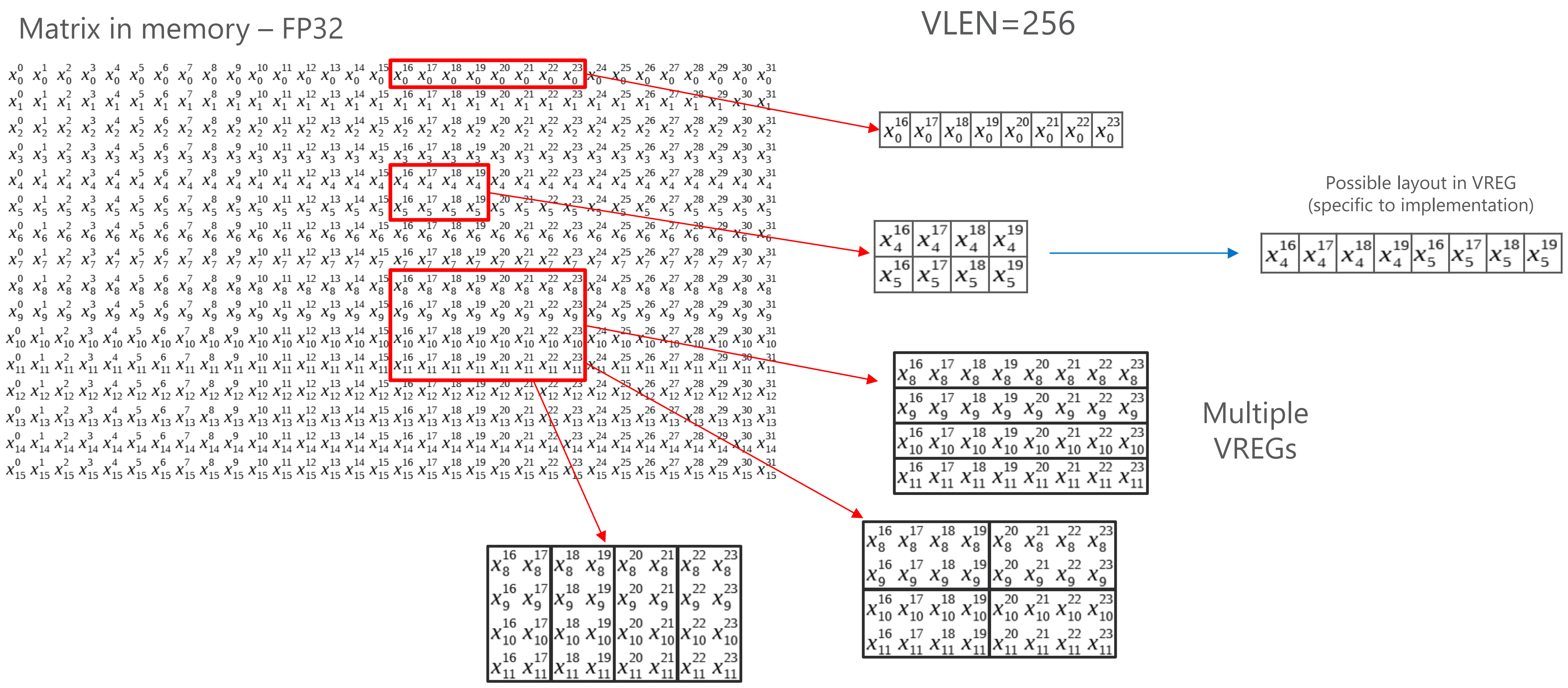
VLEN=256

Possible layout in VREG
(specific to implementation)

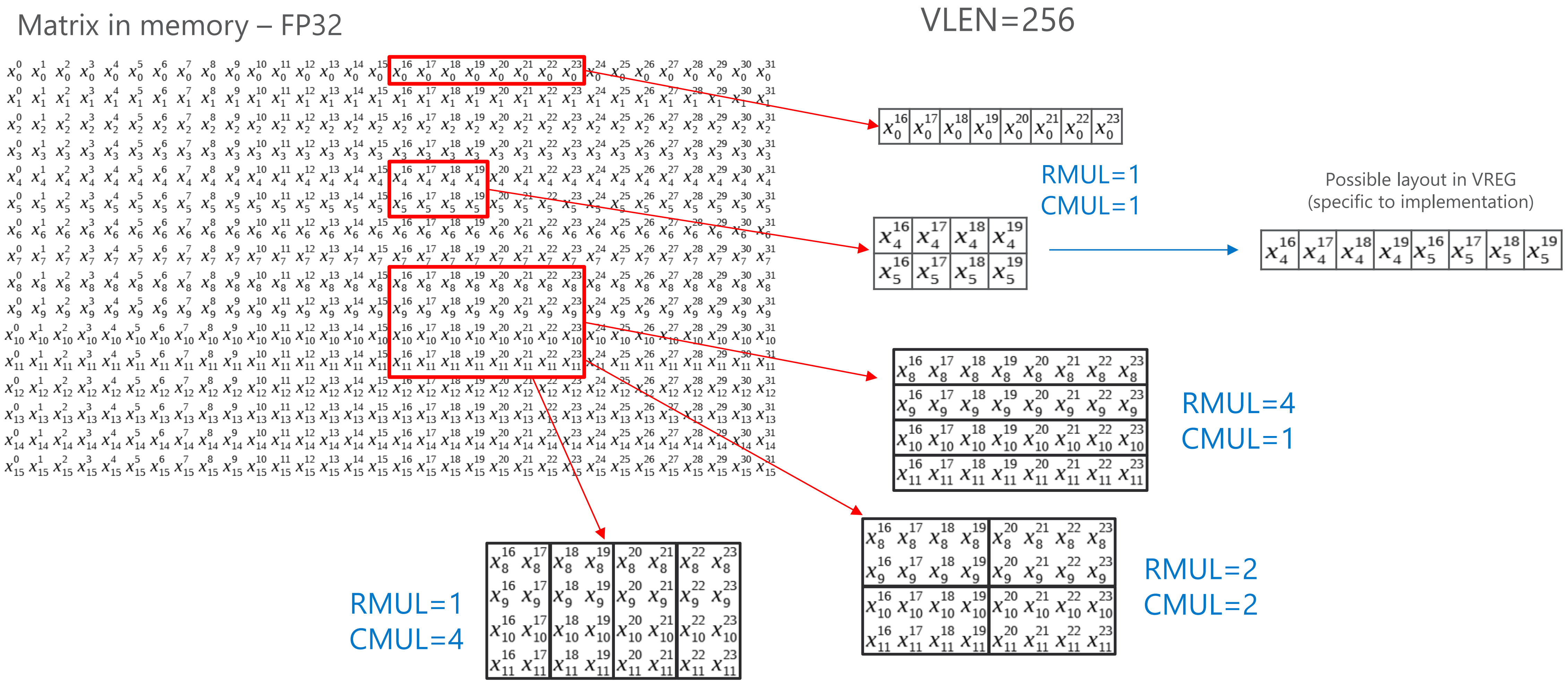


x_0^{16} x_0^{17} x_0^{18} x_0^{19} x_0^{20} x_0^{21} x_0^{22} x_0^{23}
x_4^{16} x_4^{17} x_4^{18} x_4^{19}
x_5^{16} x_5^{17} x_5^{18} x_5^{19}
x_4^{16} x_4^{17} x_4^{18} x_4^{19} x_5^{16} x_5^{17} x_5^{18} x_5^{19}

MEMORY TO VREG TILE CONTENT



MEMORY TO VREG TILE CONTENT

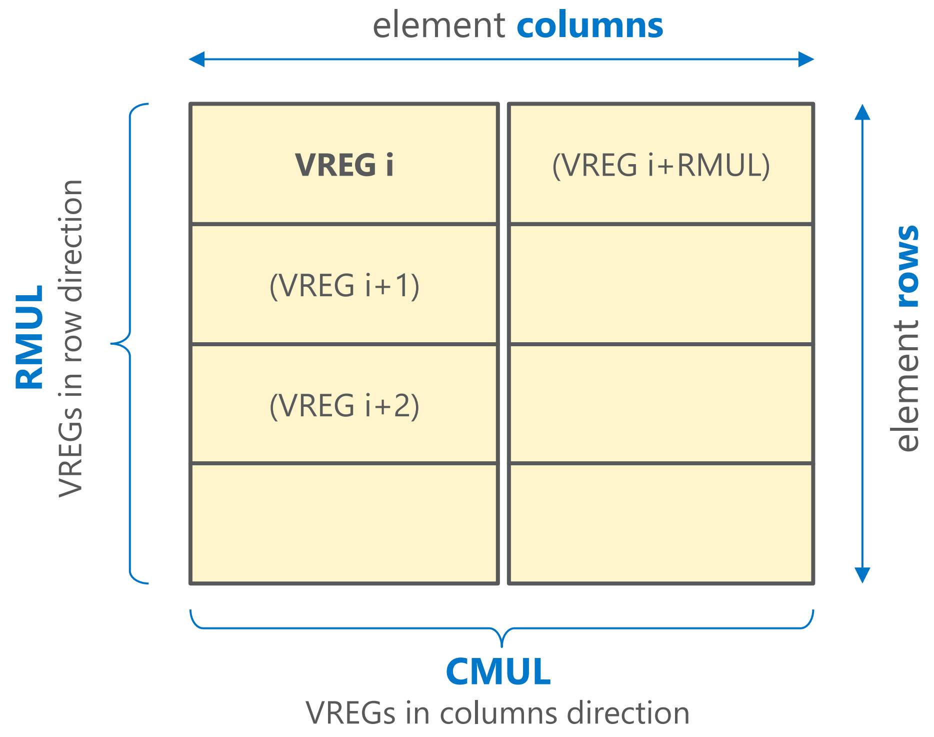


Vector Register Matrix Tiles

POSSIBLE CONVENTION FOR VREG TILES

A matrix tile is stored in a vector register or a vector register group.

- RMUL (row multiplier) stacks VREGS vertically. Power of 2. Maximum 16.
- CMUL (column multiplier) multiplies number of VREG columns. Power of 2. Maximum 16.
- Specify only ***VREG i*** as argument to matrix unit commands.
- VREG group content metadata: „tile shape“



```
// example
struct TileShape {
  unsigned int rows  : MM;
  unsigned int cols  : NN;
  unsigned int rmul  :  3;
  unsigned int cmul  :  3;
  unsigned int sew   :  3;
  unsigned int mtype :  2;
  unsigned int cmo   :  1;
};
```

INSTRUCTIONS OVERVIEW

Logical instructions

GET Tile Shape

```
vmts rd, rs1, sew, mtype, order
```

LOAD Tile

```
vmlt vd, (rs2), rs1, ts
```

MULTIPLY Tiles

vmmacc.vv v_C, v_A, v_B, ts_C, ts_A, ts_B

STORE Tile

```
vmst vd, (rs2), rs1, ts
```

INSTRUCTIONS OVERVIEW

`vmts rd, rs1, sew, mtype, order`	
GET supported tile shape	rd→tile shape, rs1→requested rows, cols
`vmmset tc, ta, tb`	
SET/configure matrix multiplier	ta, tb, tc : tile shapes of A, B, C
`vmlt vd, (rs2), rs1`	
LOAD matrix tile from memory	vd→VREG grp, rs2→base address, rs1→tile type, strides,…
`vmst vd, (rs2), rs1`	
STORE matrix tile to memory	vd→VREG grp, rs2→base address, rs1→tile type, strides,…
`vmmacc.vv vd, vs1, vs2`	
matrix **MULTIPLY** and accumulate	vd→C, vs1→A, vs2→B

openchip
Enabling the Digital World

THE FLOW

QUERY

SW can request impl. limits

- max RMUL, CMUL for given SEW, matrix type, ordering, widening
- Instruction or runtime mechanism
- No effect on HW, read-only
- Assume that for RMUL, CMUL all lower values are supported, too. 4 → 2, 1

EXECUTION PLAN

SW determines appropriate settings for execution strategy

- Register allocation
- Memory hierarchy
- Prefetching or eager loading ahead of use
- Can be IF blocks selecting among multiple optimized implementations.

CONFIGURE

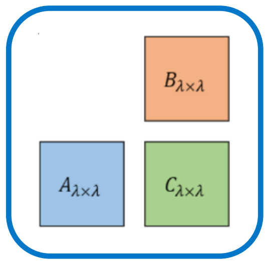
Write values corresponding to the execution plan e.g. into CSRs, configuring the matrix unit

EXECUTE

MAPPING TO IME OPTION A

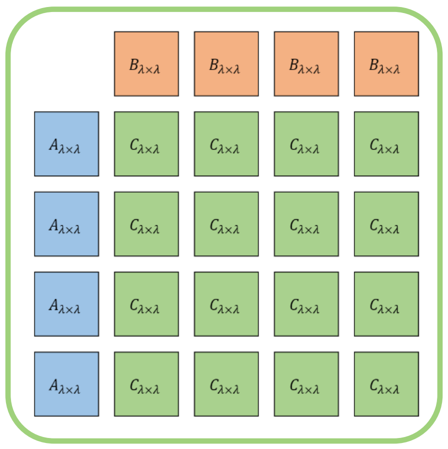
Query tile shapes

In hardware


$B_{\lambda\times\lambda}$
$A_{\lambda\times\lambda}$
$C_{\lambda\times\lambda}$

mtype	rows	cols	RMUL	CMUL
C	λ	λ	1	1
A	λ	λ	1	1
B	λ	λ	1	1

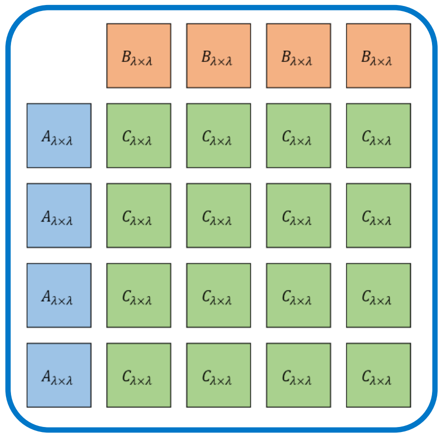
In software


$B_{\lambda\times\lambda}$ $B_{\lambda\times\lambda}$ $B_{\lambda\times\lambda}$ $B_{\lambda\times\lambda}$
$A_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$
$A_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$
$A_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$
$A_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$

One step multiply
or
multiple steps multiply

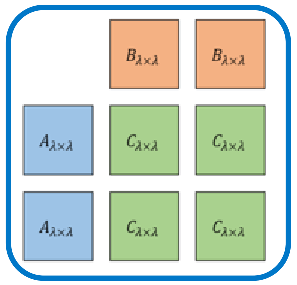
mtype	rows	cols	RMUL	CMUL
C	4λ	4λ	4	4
A	4λ	λ	4	1
B	λ	4λ	1	4

In hardware


$B_{\lambda\times\lambda}$ $B_{\lambda\times\lambda}$ $B_{\lambda\times\lambda}$ $B_{\lambda\times\lambda}$
$A_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$
$A_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$
$A_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$
$A_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$

One step multiply

In hardware


$B_{\lambda\times\lambda}$ $B_{\lambda\times\lambda}$
$A_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$
$A_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$ $C_{\lambda\times\lambda}$

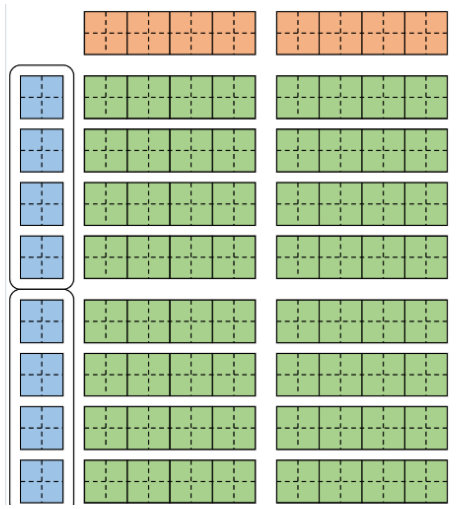
On out-of-order CPUs

On in-order CPU: use free VREGs to prefetch A, B.

mtype	rows	cols	RMUL	CMUL
C	2λ	2λ	2	2
A	2λ	λ	2	1
B	λ	2λ	1	2

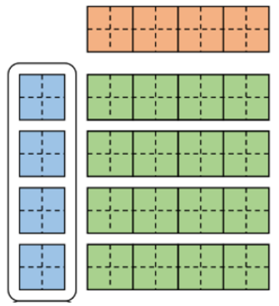
openchip Enabling the Digital World

MAPPING TO IME OPTION C

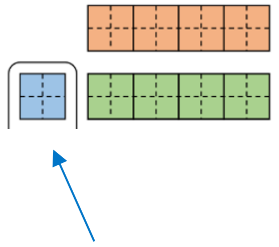


mtype	rows	cols	RMUL	CMUL
C	16	16	8	2
A	16	2	2	1
B	2	16	1	2

One step multiply



mtype	rows	cols	RMUL	CMUL
C	8	8	4	1
A	8	2	1	1
B	2	8	1	1

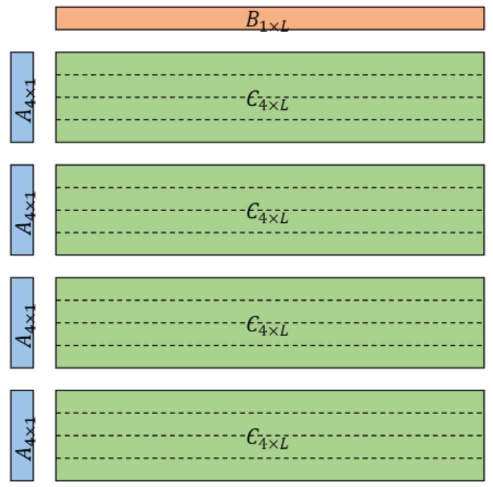


mtype	rows	cols	RMUL	CMUL
C	8	8	1	1
A	2	2	1	1
B	2	8	1	1

Using only a segment of A!

One step multiply
or
4 steps multiply

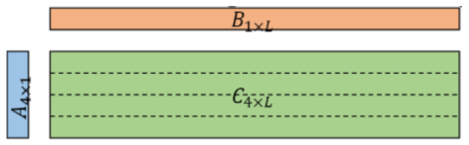
MAPPING TO IME OPTION B



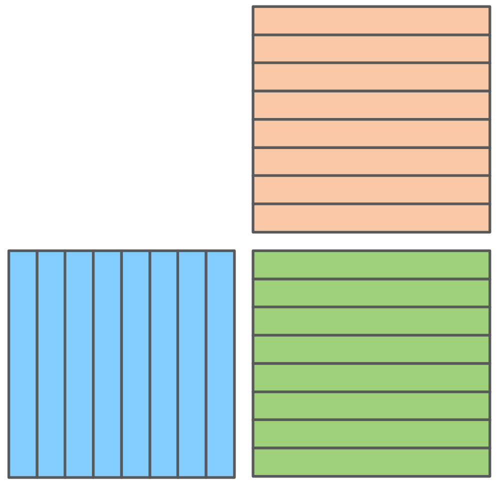
mtype	rows	cols	RMUL	CMUL
C	16	L	16	1
A	16	1	1	1
B	1	L	1	1

1 multiply

mtype	rows	cols	RMUL	CMUL
C	4	L	4	1
A	4	1	1	1
B	1	L	1	1

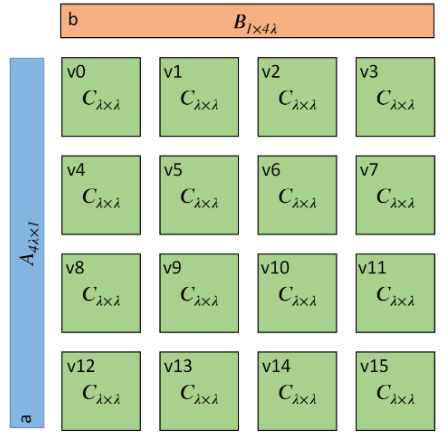


… 4 multiplies

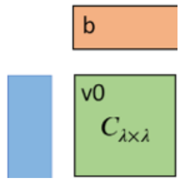


Stacking several A and B registers seems more efficient.

MAPPING TO IME OPTION D

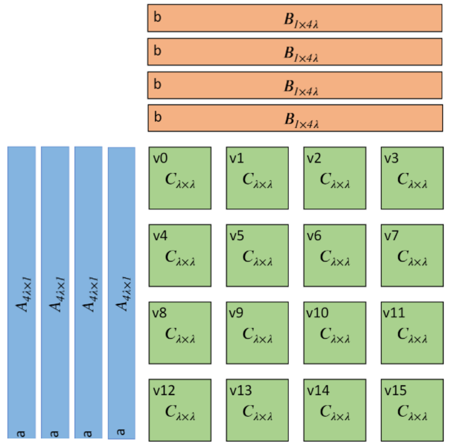

b
$B_{l\times 4\lambda}$
$A_{4\lambda\times l}$
a
v0 $C_{\lambda\times\lambda}$
v1 $C_{\lambda\times\lambda}$
v2 $C_{\lambda\times\lambda}$
v3 $C_{\lambda\times\lambda}$
v4 $C_{\lambda\times\lambda}$
v5 $C_{\lambda\times\lambda}$
v6 $C_{\lambda\times\lambda}$
v7 $C_{\lambda\times\lambda}$
v8 $C_{\lambda\times\lambda}$
v9 $C_{\lambda\times\lambda}$
v10 $C_{\lambda\times\lambda}$
v11 $C_{\lambda\times\lambda}$
v12 $C_{\lambda\times\lambda}$
v13 $C_{\lambda\times\lambda}$
v14 $C_{\lambda\times\lambda}$
v15 $C_{\lambda\times\lambda}$

mtype	rows	cols	RMUL	CMUL
C	4λ	4λ	4	4
A	4λ	1	1	1
B	1	4λ	1	1



a bit wasteful...

a bit better?

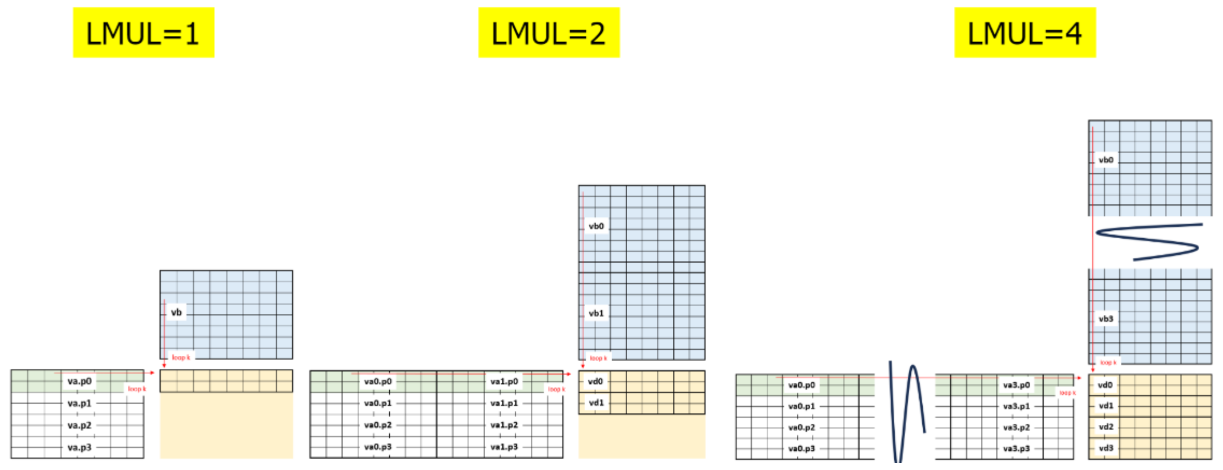

b $B_{l\times 4\lambda}$
b $B_{l\times 4\lambda}$
b $B_{l\times 4\lambda}$
b $B_{l\times 4\lambda}$
$A_{4\lambda\times l}$ a
$A_{4\lambda\times l}$ a
$A_{4\lambda\times l}$ a
$A_{4\lambda\times l}$ a
v0 $C_{\lambda\times\lambda}$
v1 $C_{\lambda\times\lambda}$
v2 $C_{\lambda\times\lambda}$
v3 $C_{\lambda\times\lambda}$
v4 $C_{\lambda\times\lambda}$
v5 $C_{\lambda\times\lambda}$
v6 $C_{\lambda\times\lambda}$
v7 $C_{\lambda\times\lambda}$
v8 $C_{\lambda\times\lambda}$
v9 $C_{\lambda\times\lambda}$
v10 $C_{\lambda\times\lambda}$
v11 $C_{\lambda\times\lambda}$
v12 $C_{\lambda\times\lambda}$
v13 $C_{\lambda\times\lambda}$
v14 $C_{\lambda\times\lambda}$
v15 $C_{\lambda\times\lambda}$

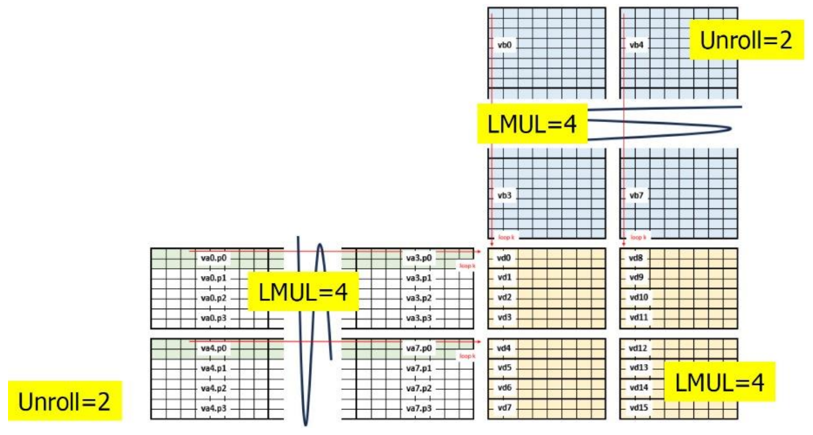
mtype	rows	cols	RMUL	CMUL
C	4λ	4λ	4	4
A	4λ	1	1	4
B	1	4λ	4	1

This implementation option is excellent for column-major-order A and row-major-order B (not what the evaluation was asking for...)

openchip Enabling the Digital World

MAPPING IME OPTION E



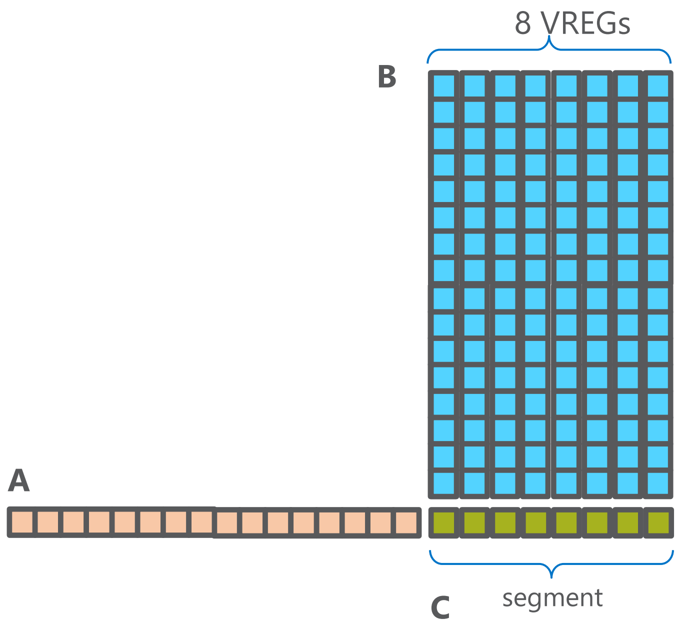


mtype	rows	cols	RMUL	CMUL
C	8	8	4	1
A	8	32	1	4
B	32	8	4	1

mtype	rows	cols	RMUL	CMUL
C	16	16	8	2
A	16	32	2	4
B	32	16	4	2

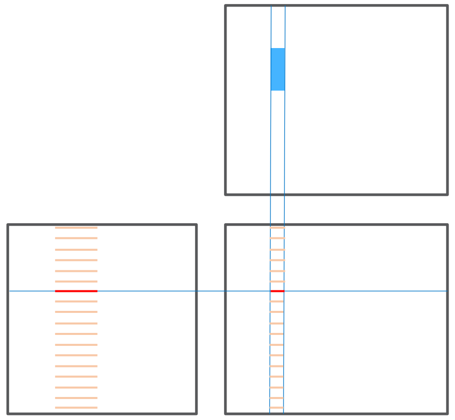
openchip
Enabling the Digital World

MAPPING IME OPTION F



mtype	rows	cols	RMUL	CMUL	order
C	1	8	1	1	rmo
A	1	VLEN/SEW	1	1	rmo
B	VLEN/SEW	8	8	1	cmo

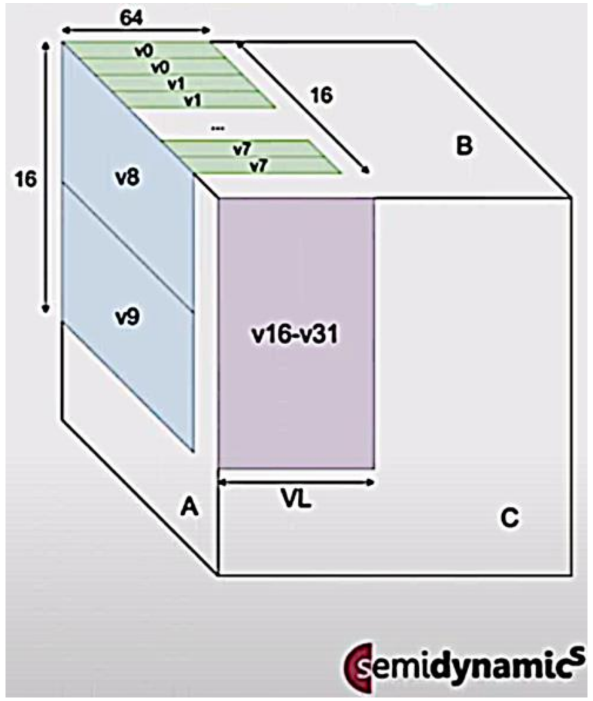
Straight forward …



Segment handling needs vslideup or similar …
But is it really needed if B is weight stationary?

NOTE: a **weight stationary** implementation needs a different order of strip-mining loops! N, K, M

IME TG - SEMIDYNAMICS

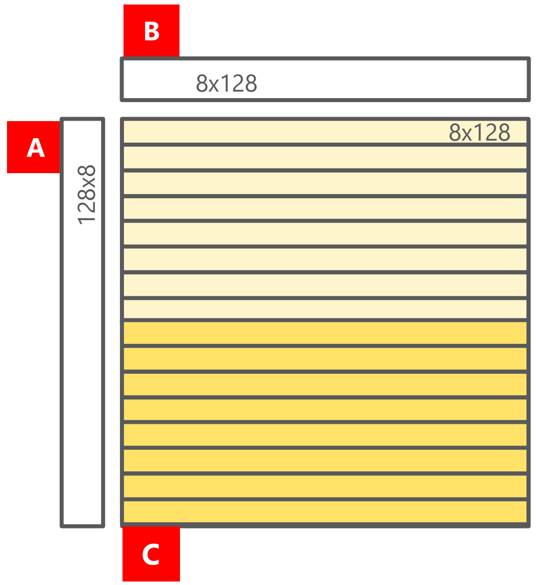


mtype	rows	cols	RMUL	CMUL
C	16	64	16	1
A	16	16	2	1
B	16	64	8	1

Straight forward …

COLUMN-MAJOR-ORDER A MATRIX – HPC DESIGN POINT

FP16, INT16
widen 1x



mtype	rows	cols	RMUL	CMUL
C	128	128	16	1
A	128	8	1	1
B	8	128	1	1

QUERY MATRIX TILE SHAPE

vmts
vfmts
vfwmts
vfqmts

New instruction: Vector Float Matrix Tile Shape:

vmts rd, rs1, sew, mtype, order

Retrieve the Tile Shape supported by the matrix extension for a particular tile type A, B or C which matches the desired maximum rows/columns. Equivalent to *vsetvl*.

```
// example
struct TileShape {
  unsigned int rows  : MM;
  unsigned int cols  : NN;
  unsigned int rmul  :  3;
  unsigned int cmul  :  3;
  unsigned int sew   :  3;
  unsigned int mtype :  2;
  unsigned int cmo   :  1;
};
```

Arguments:

- **rd:** destination scalar register, contains tile shape result
- **rs1:** source register contains desired (maximum) rows, columns shape in memory (16 bit each, for example)
- **sew:** desired single element width, like vsetvl: *e8, e16, e32, e64*
- **mtype:** matrix type for A, B, C type: *mta, mtb, mtc*
- **order:** order in memory, column- or row-major-order: *cmo, rmo*

Returns:

- A TileShape value that is supported by the matrix unit implementation.

openchip
Enabling the Digital World

MATRIX TILE SHAPE – MORE DETAILS

Program needs to multiply matrices: C=A*B
Example: C: 512x256, A: 512x1024, B: 1024x256 (#rows x #columns of elements)

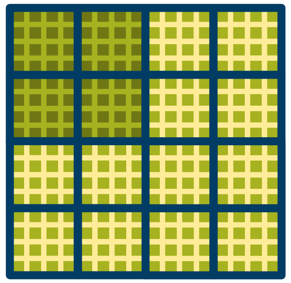
- Ask multiplier for geometry information
  t1 ← (256 << 16) | 512
  ```
  vmts tc, t1, e32, mtc, rmo
  ```

- The multiplier replies with a tile shape stored in the scalar register *tc*.
  For example, on a VLEN=512 system:
  tc = {rows=16, cols=16, rmul=4, cmul=4, sew=32, mtype=mtc, cmo=0}

  - If requested geometry is smaller than the supported size but can be processed by the multiplier (e.g. by filling in zeros): rows, cols shall correspond to the asked geometry.
  - If the multiplier does not support the smaller geometry, return zero. → process in vector unit.

- We use tc to load a piece of matrix into the vector register group.
  - The exact order of elements in the vector registers is not prescribed. It is left to the implementation.
  - Portability can only be achieved with implementation specific tile load/store instructions.

```
// example
struct TileShape {
  unsigned int rows  : 13;
  unsigned int cols  : 13;
  unsigned int rmul  :  3;
  unsigned int cmul  :  3;
  unsigned int sew   :  3;
  unsigned int mtype :  2;
  unsigned int cmo   :  1;
};
```



WHAT ABOUT THE ACCUMULATOR?

✓ Case 1: Store directly to vector registers.
  - Simple and easy, more cycles and data movement.
  - OK for short vectors, small accumulators

✓ Case 2: treat accumulator like register bypass

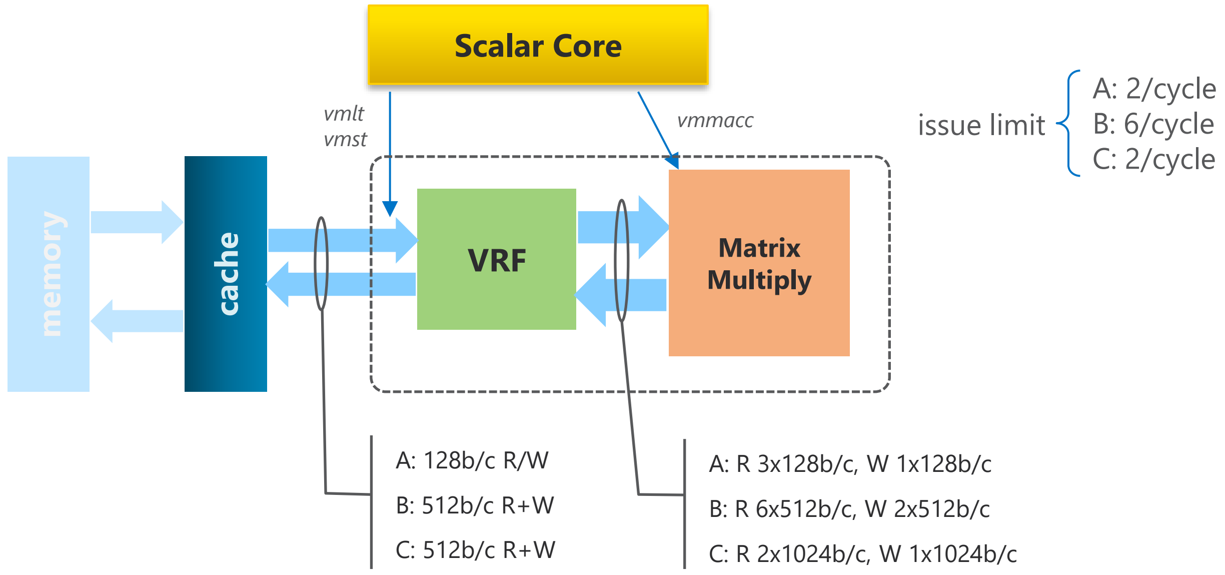
✓ **Register renaming**
- For *vmmacc* ***vd*** is input & output register!
- No commit until next instruction uses register
- IF instruction is *vmmacc*
  - IF previous instruction was not *vmmacc*
    - Read *vd* into *Acc*
- ELSE IF previous instruction is *vmmacc*
  - Write *Acc* to *vd*

✓ **"Reservation Station"**
- Matrix multiply using *vd* lands in R.S.
  - Once *vd* is available: transfers data into *acc*
- When matrix multiply is finished:
  vmmacc remains in R.S.
  - Retired when subsequent *vmmacc* is issued, reusing the *vd*
  - If another instruction is issued which touches *vd*:
    - stores *acc* to *vd*
    - retires

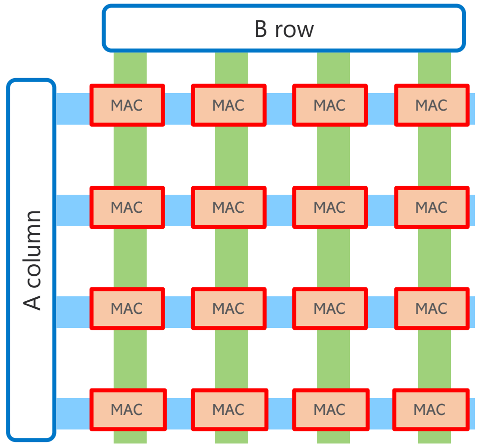
RISC-V IME TG EVALUATION SCENARIOS

- ✓ A: Low-end “embedded” core
  - VLEN = 128, DLEN = 128
  - # of RF rd/wr ports = 3Rd + 1Wr, possibly 2-way banked (plus Rd/Wr ports for St/Ld)
  - # of cache rd/wr accesses = 1Rd/Wr, VLEN-wide
  - In-order, two-wide superscalar, no register renaming, ~2 GHz
- ✓ B: High-end “apps” core
  - VLEN = 512, DLEN = 512 (or 2x256)
  - # of RF rd/wr ports = 6Rd + 2Wr, possibly 2-way banked (plus Rd/Wr ports for St/Ld)
  - # of cache rd/wr accesses = 1Rd + 1Wr, VLEN-wide (or 2x VLEN/2)
  - Out-of-order, six-wide superscalar, full register renaming (speculative execution), ~3 GHz
- ✓ C: “HPC” core
  - VLEN = 16K, DLEN = 1K (as 16 64-bit-wide “lanes”)
  - # of RF rd/wr ports = 32 8B-wide Rd + 16 8B-wide Wr
  - # of cache rd/wr accesses = 1Rd + 1Wr, DLEN/2=512b-wide
  - Limited out-of-order, two-wide superscalar, limited register renaming (speculative execution), ~2 GHz

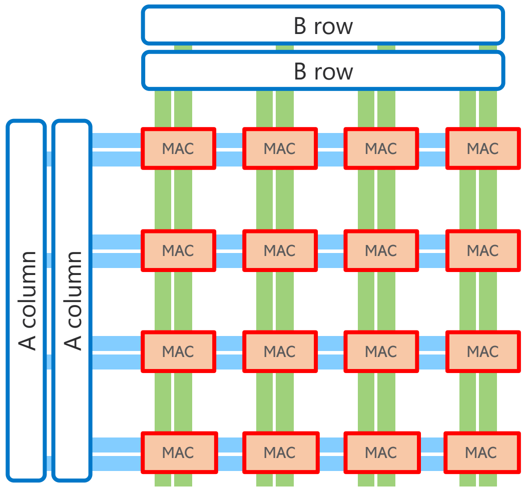
CONSTRAINTS



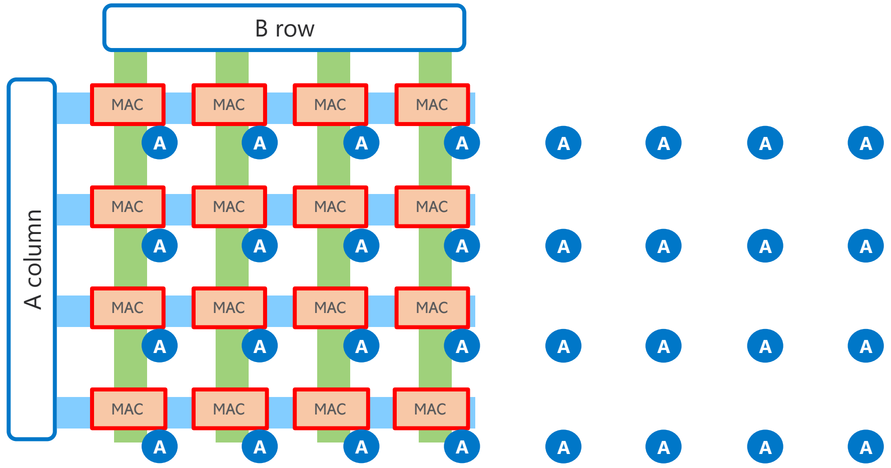
OUTER PRODUCT



Outer product
Rank-1 update

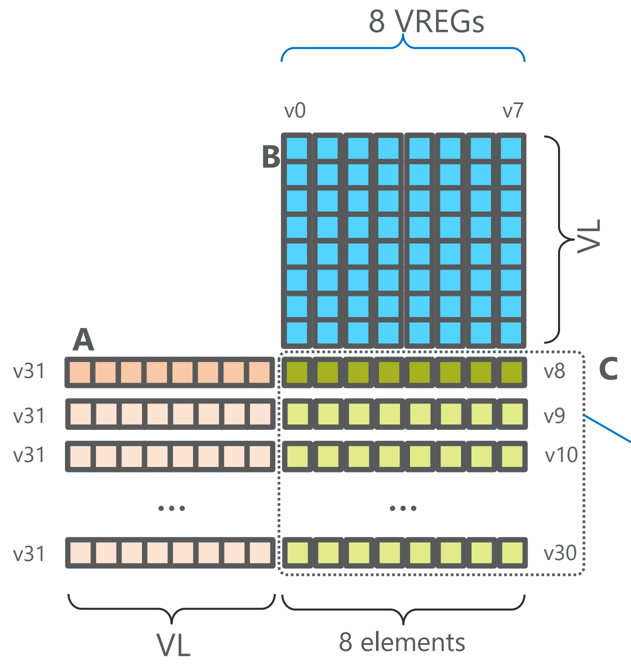


Outer product
Rank-2 update

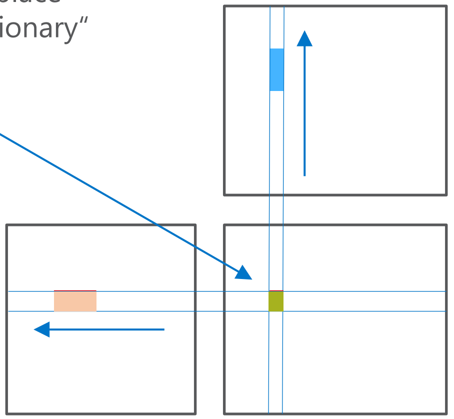


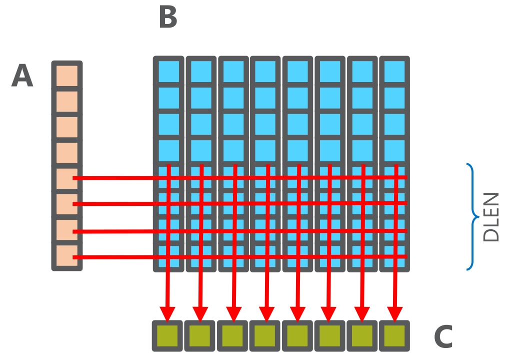
Outer product
Accumulator larger than physical MAC array

INNER PRODUCT



C is kept in place
„output stationary“

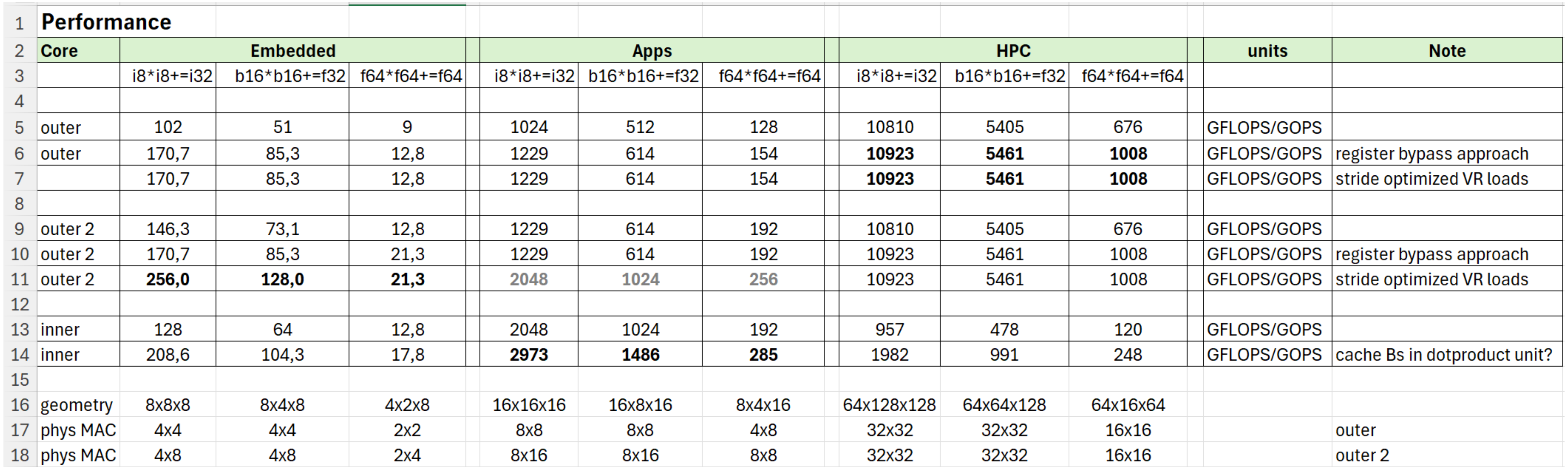




8*DLEN/SEW multiplies / cycle

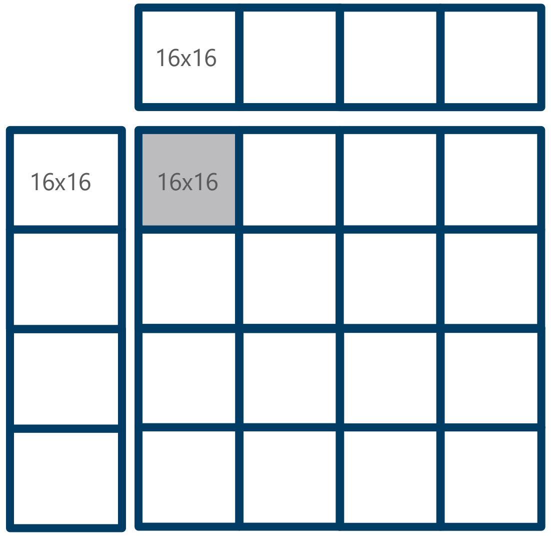
Outer product with BW DLEN/cycle:
(DLEN/(2*SEW))^2 multiplies / cycle
- Can win if using full load BW for A,B
- If DLEN/SEW ≥ 32

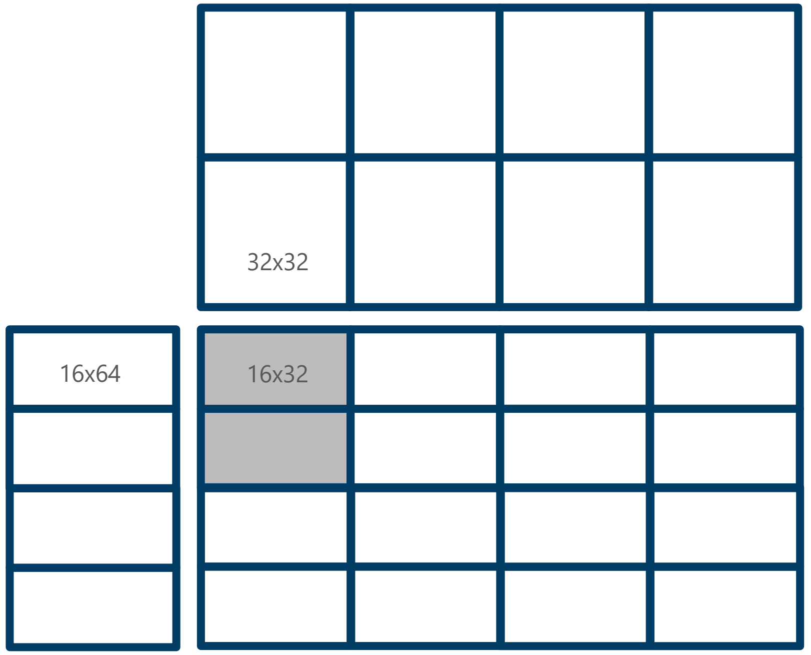
OVERVIEW INNER VS. OUTER PRODUCT PERFORMANCE

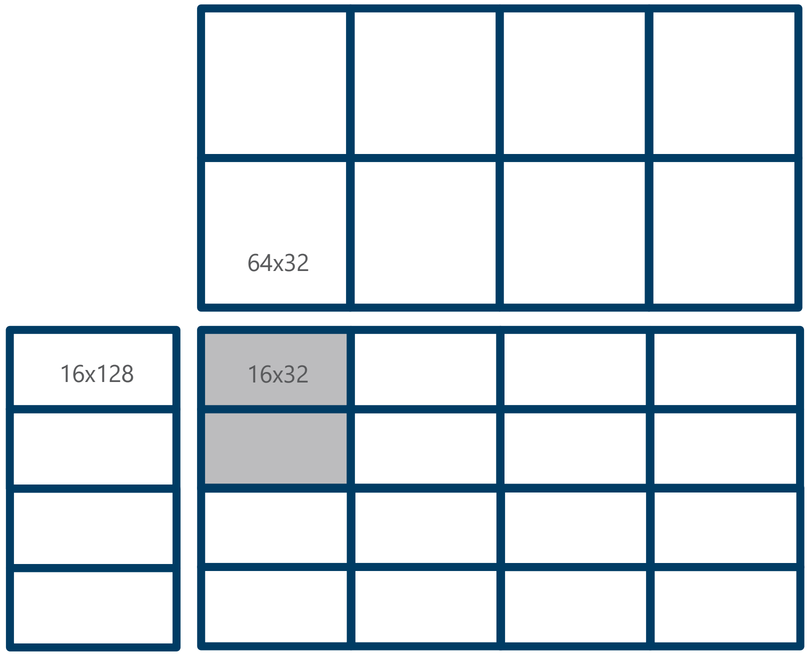


1	**Performance**											
2	**Core**	**Embedded**			**Apps**			**HPC**			**units**	**Note**
3		i8*i8+=i32	b16*b16+=f32	f64*f64+=f64	i8*i8+=i32	b16*b16+=f32	f64*f64+=f64	i8*i8+=i32	b16*b16+=f32	f64*f64+=f64		
4												
5	outer	102	51	9	1024	512	128	10810	5405	676	GFLOPS/GOPS	
6	outer	170,7	85,3	12,8	1229	614	154	**10923**	**5461**	**1008**	GFLOPS/GOPS	register bypass approach
7		170,7	85,3	12,8	1229	614	154	**10923**	**5461**	**1008**	GFLOPS/GOPS	stride optimized VR loads
8												
9	outer 2	146,3	73,1	12,8	1229	614	192	10810	5405	676	GFLOPS/GOPS	
10	outer 2	170,7	85,3	21,3	1229	614	192	10923	5461	1008	GFLOPS/GOPS	register bypass approach
11	outer 2	**256,0**	**128,0**	**21,3**	**2048**	**1024**	**256**	10923	5461	1008	GFLOPS/GOPS	stride optimized VR loads
12												
13	inner	128	64	12,8	2048	1024	192	957	478	120	GFLOPS/GOPS	
14	inner	208,6	104,3	17,8	**2973**	**1486**	**285**	1982	991	248	GFLOPS/GOPS	cache Bs in dotproduct unit?
15												
16	geometry	8x8x8	8x4x8	4x2x8	16x16x16	16x8x16	8x4x16	64x128x128	64x64x128	64x16x64		
17	phys MAC	4x4	4x4	2x2	8x8	8x8	4x8	32x32	32x32	16x16		outer
18	phys MAC	4x8	4x8	2x4	8x16	8x16	8x8	32x32	32x32	16x16		outer 2

„HPC“ CORE







mtype	rows	cols	RMUL	CMUL
C	64	64	4	4
A	64	16	4	1
B	16	64	1	4

mtype	rows	cols	RMUL	CMUL
C	64	128	4	4
A	64	64	4	1
B	64	128	2	4

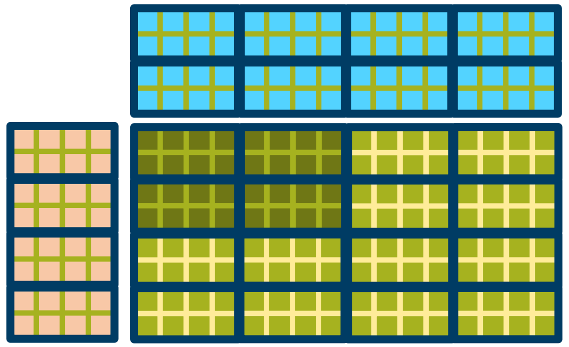
mtype	rows	cols	RMUL	CMUL
C	64	128	4	4
A	64	128	4	1
B	128	128	2	4

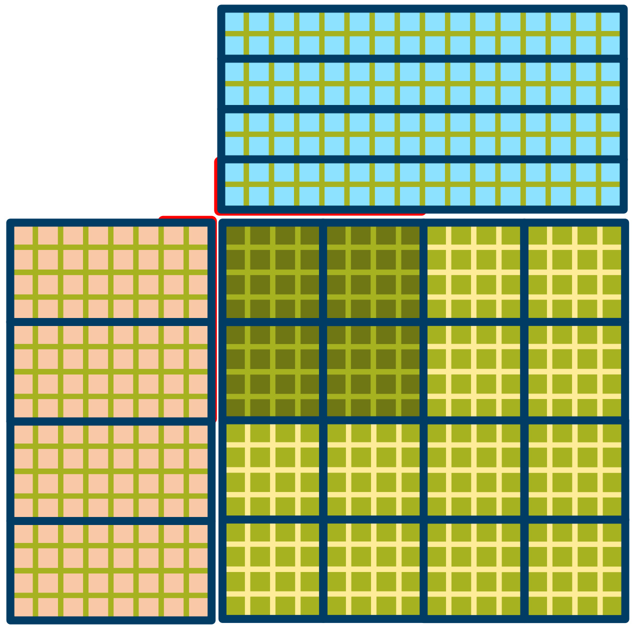
fp64 += fp64*fp64

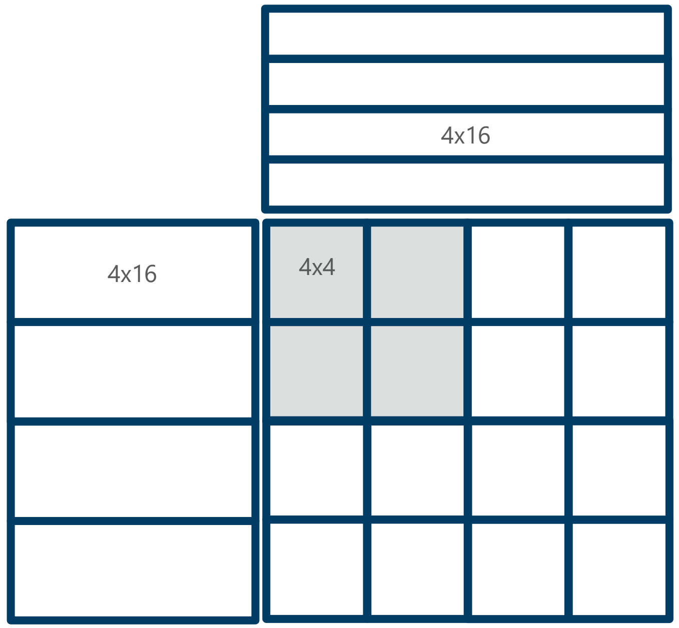
fp32 += bf16*bf16

int32 += int8*int8

HIGH-END „APPS“ CORE







mtype	rows	cols	RMUL	CMUL
C	8	16	4	4
A	8	4	4	1
B	4	16	2	4

mtype	rows	cols	RMUL	CMUL
C	16	16	4	4
A	16	8	4	1
B	8	16	4	1

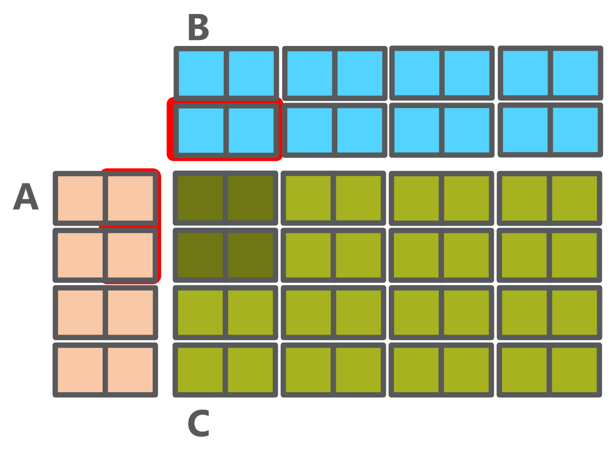
mtype	rows	cols	RMUL	CMUL
C	16	16	4	4
A	16	16	4	1
B	16	16	4	1

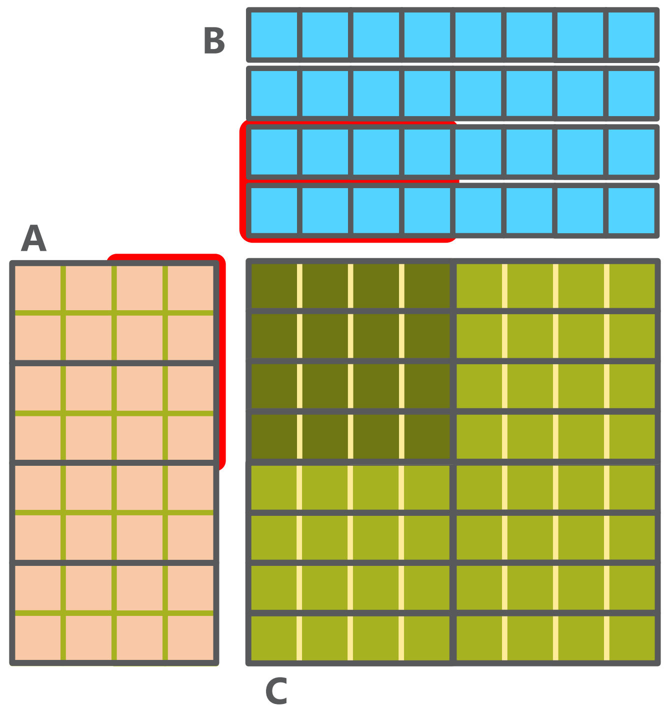
fp64 += fp64*fp64

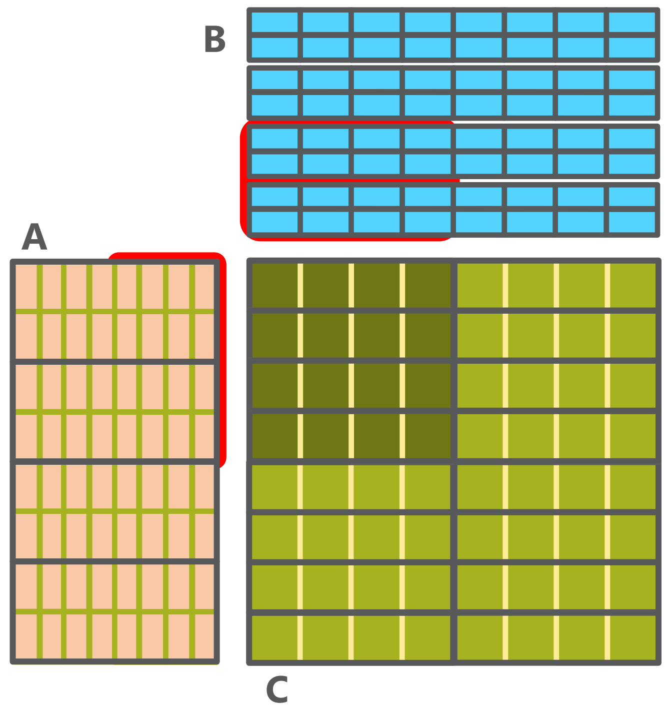
fp32 += bf16*bf16

int32 += int8*int8

LOW-END „EMBEDDED“ CORE







mtype	rows	cols	RMUL	CMUL
C	4	8	4	4
A	4	2	4	1
B	2	8	2	4

fp64 += fp64*fp64

mtype	rows	cols	RMUL	CMUL
C	8	8	8	2
A	8	4	4	1
B	4	8	4	1

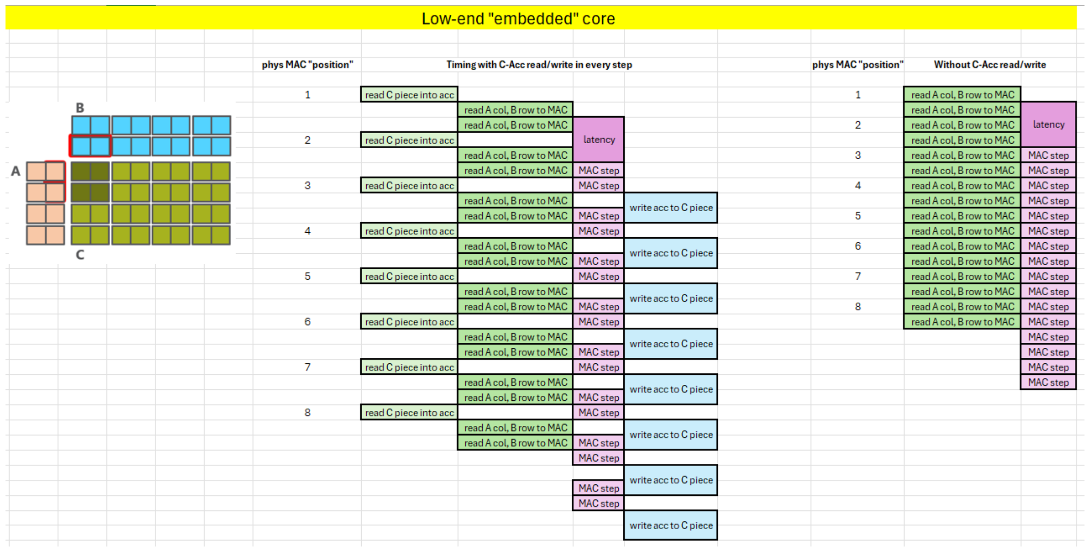
fp32 += bf16*bf16

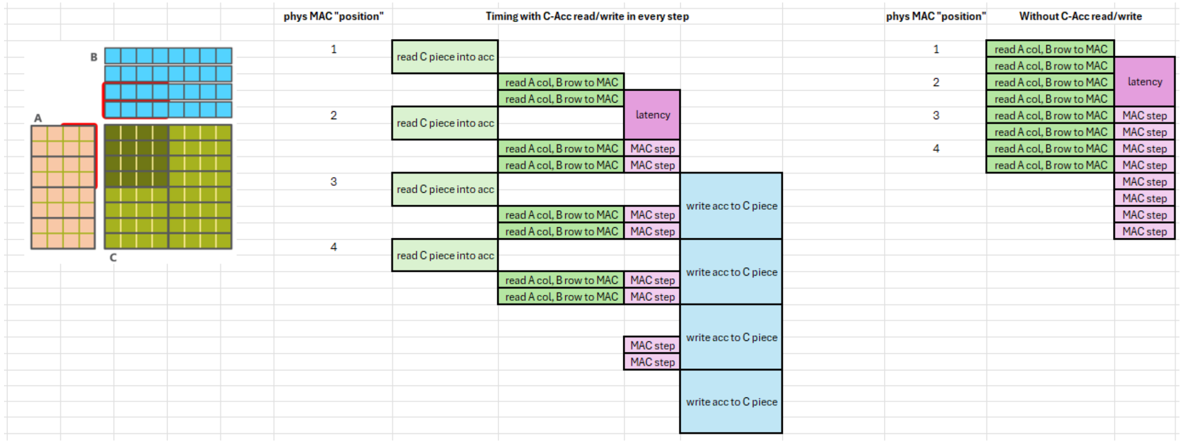
mtype	rows	cols	RMUL	CMUL
C	8	8	8	2
A	4	2	4	1
B	2	8	2	4

int32 += int8*int8

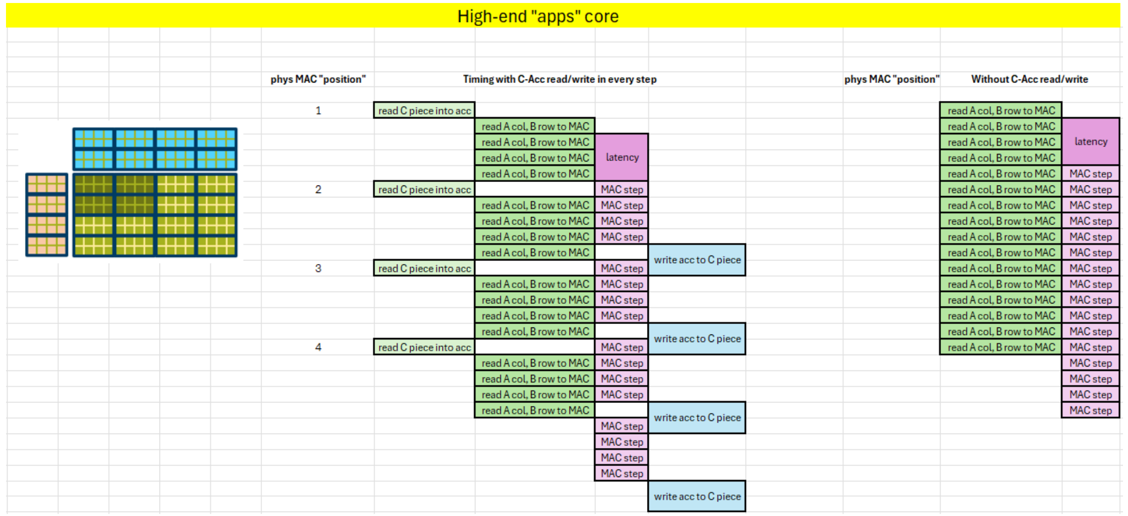
EVALUATION - PIPELINES

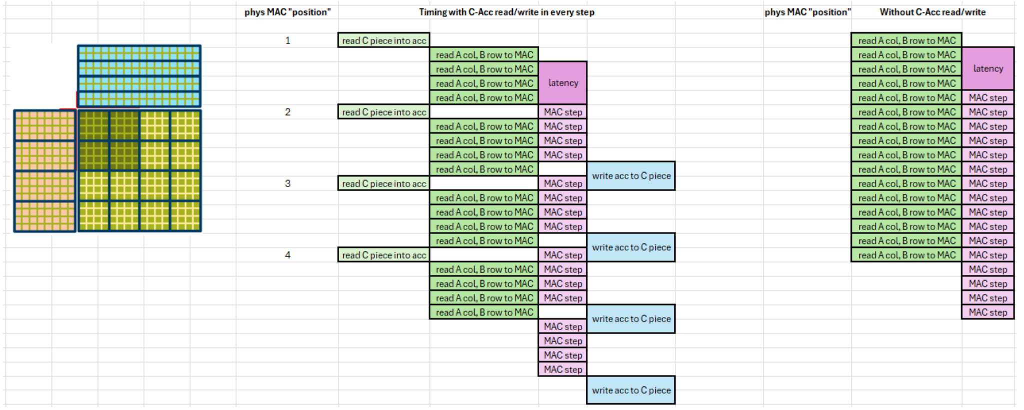
Low-end "embedded" core



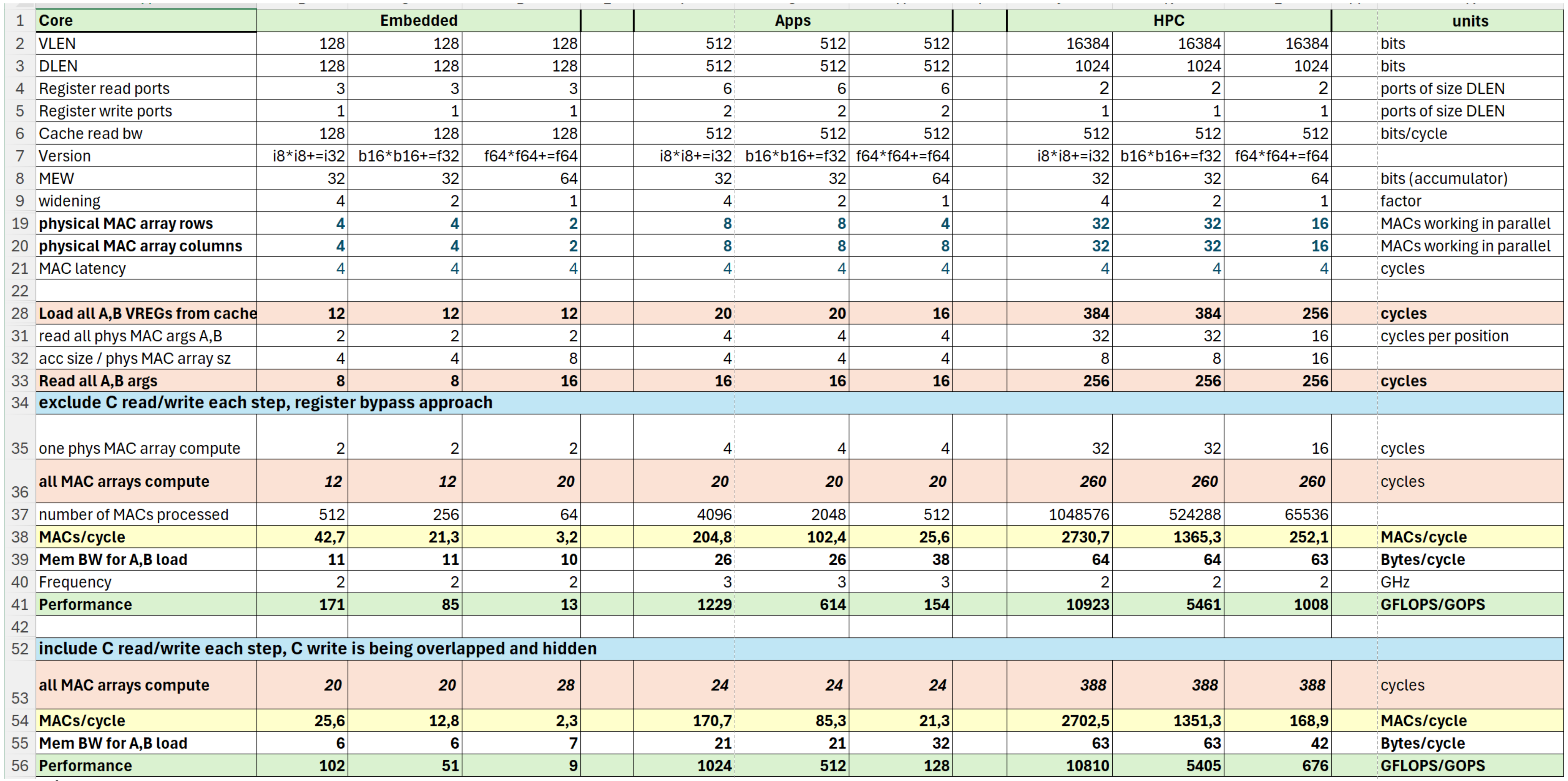


High-end "apps" core



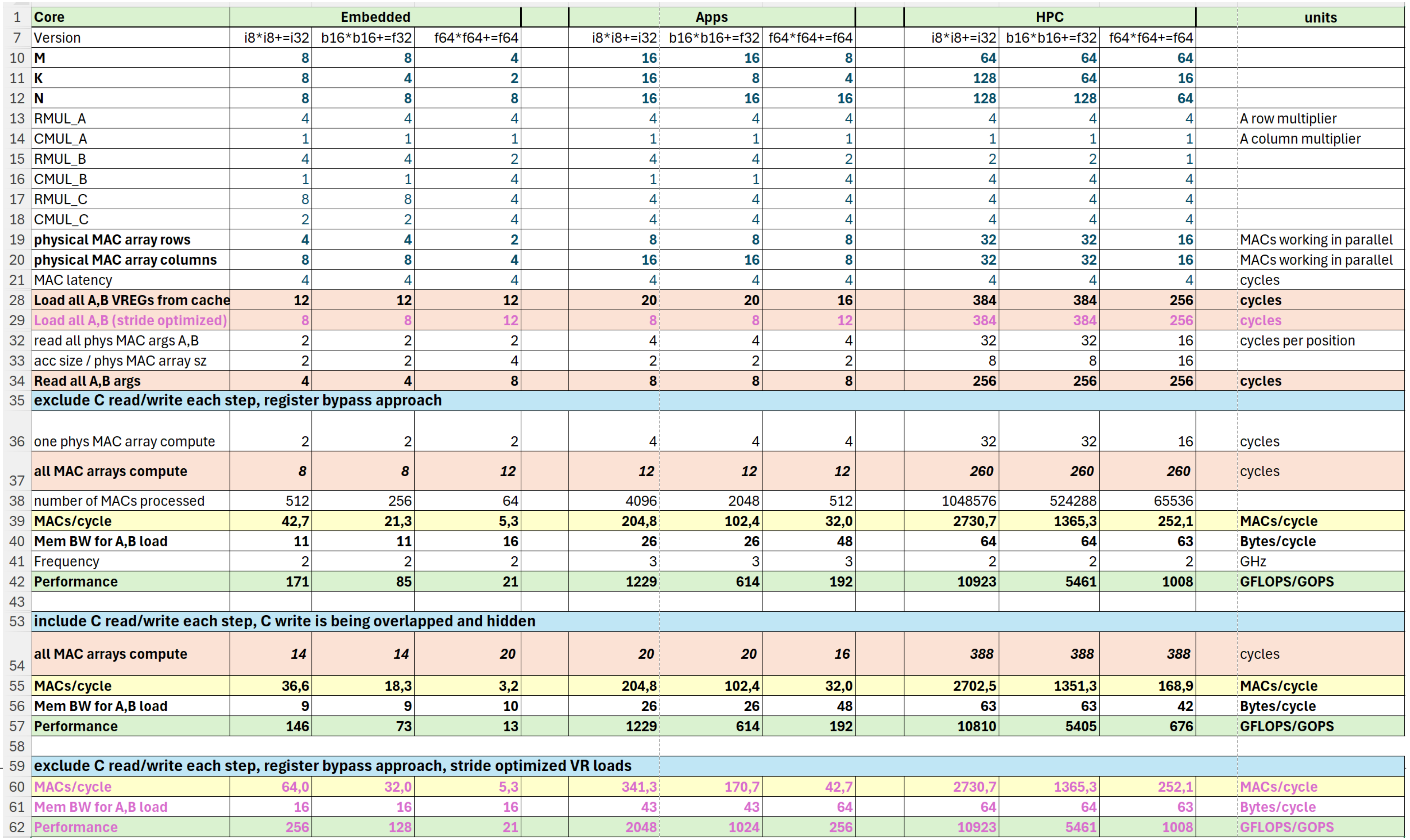


Evaluation – Outer Product



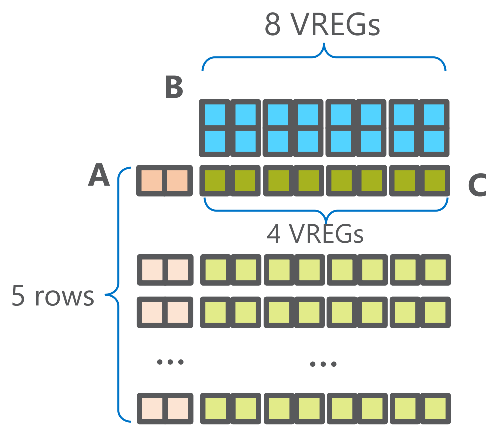
	Core	Embedded				Apps				HPC				units
2	VLEN	128	128	128		512	512	512		16384	16384	16384		bits
3	DLEN	128	128	128		512	512	512		1024	1024	1024		bits
4	Register read ports	3	3	3		6	6	6		2	2	2		ports of size DLEN
5	Register write ports	1	1	1		2	2	2		1	1	1		ports of size DLEN
6	Cache read bw	128	128	128		512	512	512		512	512	512		bits/cycle
7	Version	i8*i8+=i32	b16*b16+=f32	f64*f64+=f64		i8*i8+=i32	b16*b16+=f32	f64*f64+=f64		i8*i8+=i32	b16*b16+=f32	f64*f64+=f64		
8	MEW	32	32	64		32	32	64		32	32	64		bits (accumulator)
9	widening	4	2	1		4	2	1		4	2	1		factor
19	**physical MAC array rows**	**4**	**4**	**2**		**8**	**8**	**4**		**32**	**32**	**16**		MACs working in parallel
20	**physical MAC array columns**	**4**	**4**	**2**		**8**	**8**	**8**		**32**	**32**	**16**		MACs working in parallel
21	MAC latency	4	4	4		4	4	4		4	4	4		cycles
22														
28	**Load all A,B VREGs from cache**	**12**	**12**	**12**		**20**	**20**	**16**		**384**	**384**	**256**		**cycles**
31	read all phys MAC args A,B	2	2	2		4	4	4		32	32	16		cycles per position
32	acc size / phys MAC array sz	4	4	8		4	4	4		8	8	16		
33	**Read all A,B args**	**8**	**8**	**16**		**16**	**16**	**16**		**256**	**256**	**256**		**cycles**
34	**exclude C read/write each step, register bypass approach**													
35	one phys MAC array compute	2	2	2		4	4	4		32	32	16		cycles
36	**all MAC arrays compute**	***12***	***12***	***20***		***20***	***20***	***20***		***260***	***260***	***260***		cycles
37	number of MACs processed	512	256	64		4096	2048	512		1048576	524288	65536		
38	**MACs/cycle**	**42,7**	**21,3**	**3,2**		**204,8**	**102,4**	**25,6**		**2730,7**	**1365,3**	**252,1**		**MACs/cycle**
39	**Mem BW for A,B load**	**11**	**11**	**10**		**26**	**26**	**38**		**64**	**64**	**63**		**Bytes/cycle**
40	Frequency	2	2	2		3	3	3		2	2	2		GHz
41	**Performance**	**171**	**85**	**13**		**1229**	**614**	**154**		**10923**	**5461**	**1008**		**GFLOPS/GOPS**
42														
52	**include C read/write each step, C write is being overlapped and hidden**													
53	**all MAC arrays compute**	***20***	***20***	***28***		***24***	***24***	***24***		***388***	***388***	***388***		cycles
54	**MACs/cycle**	**25,6**	**12,8**	**2,3**		**170,7**	**85,3**	**21,3**		**2702,5**	**1351,3**	**168,9**		**MACs/cycle**
55	**Mem BW for A,B load**	**6**	**6**	**7**		**21**	**21**	**32**		**63**	**63**	**42**		**Bytes/cycle**
56	**Performance**	**102**	**51**	**9**		**1024**	**512**	**128**		**10810**	**5405**	**676**		**GFLOPS/GOPS**

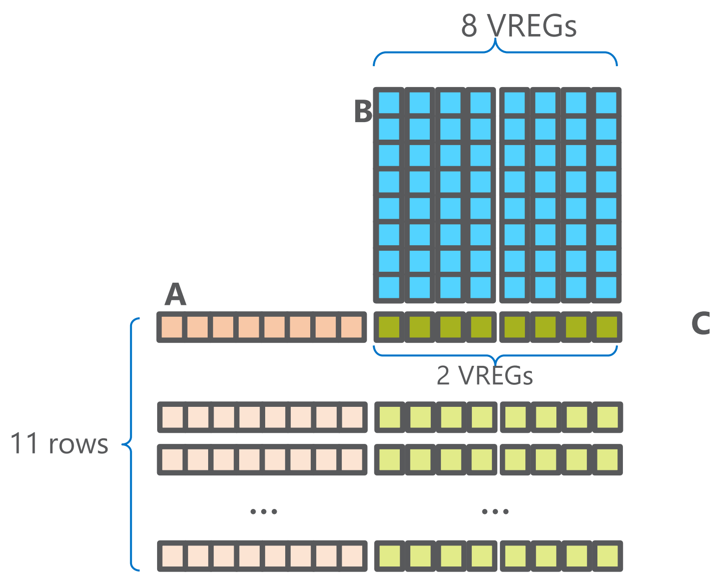
EVALUATION – OUTER PRODUCT – 2 (INCREASED PHYS MAC ARRAY)



	Core	Embedded			Apps			HPC			units
7	Version	i8*i8+=i32	b16*b16+=f32	f64*f64+=f64	i8*i8+=i32	b16*b16+=f32	f64*f64+=f64	i8*i8+=i32	b16*b16+=f32	f64*f64+=f64	
10	**M**	8	8	4	16	16	8	64	64	64	
11	**K**	8	4	2	16	8	4	128	64	16	
12	**N**	8	8	8	16	16	16	128	128	64	
13	RMUL_A	4	4	4	4	4	4	4	4	4	A row multiplier
14	CMUL_A	1	1	1	1	1	1	1	1	1	A column multiplier
15	RMUL_B	4	4	2	4	4	2	2	2	1	
16	CMUL_B	1	1	4	1	1	4	4	4	4	
17	RMUL_C	8	8	4	4	4	4	4	4	4	
18	CMUL_C	2	2	4	4	4	4	4	4	4	
19	**physical MAC array rows**	4	4	2	8	8	8	32	32	16	MACs working in parallel
20	**physical MAC array columns**	8	8	4	16	16	8	32	32	16	MACs working in parallel
21	MAC latency	4	4	4	4	4	4	4	4	4	cycles
28	**Load all A,B VREGs from cache**	**12**	**12**	**12**	**20**	**20**	**16**	**384**	**384**	**256**	**cycles**
29	**Load all A,B (stride optimized)**	8	8	12	8	8	12	384	384	256	cycles
32	read all phys MAC args A,B	2	2	2	4	4	4	32	32	16	cycles per position
33	acc size / phys MAC array sz	2	2	4	2	2	2	8	8	16	
34	**Read all A,B args**	**4**	**4**	**8**	**8**	**8**	**8**	**256**	**256**	**256**	**cycles**
35	**exclude C read/write each step, register bypass approach**										
36	one phys MAC array compute	2	2	2	4	4	4	32	32	16	cycles
37	**all MAC arrays compute**	***8***	***8***	***12***	***12***	***12***	***12***	***260***	***260***	***260***	cycles
38	number of MACs processed	512	256	64	4096	2048	512	1048576	524288	65536	
39	**MACs/cycle**	**42,7**	**21,3**	**5,3**	**204,8**	**102,4**	**32,0**	**2730,7**	**1365,3**	**252,1**	**MACs/cycle**
40	**Mem BW for A,B load**	**11**	**11**	**16**	**26**	**26**	**48**	**64**	**64**	**63**	**Bytes/cycle**
41	Frequency	2	2	2	3	3	3	2	2	2	GHz
42	**Performance**	**171**	**85**	**21**	**1229**	**614**	**192**	**10923**	**5461**	**1008**	**GFLOPS/GOPS**
43											
53	**include C read/write each step, C write is being overlapped and hidden**										
54	**all MAC arrays compute**	***14***	***14***	***20***	***20***	***20***	***16***	***388***	***388***	***388***	cycles
55	**MACs/cycle**	**36,6**	**18,3**	**3,2**	**204,8**	**102,4**	**32,0**	**2702,5**	**1351,3**	**168,9**	**MACs/cycle**
56	**Mem BW for A,B load**	**9**	**9**	**10**	**26**	**26**	**48**	**63**	**63**	**42**	**Bytes/cycle**
57	**Performance**	**146**	**73**	**13**	**1229**	**614**	**192**	**10810**	**5405**	**676**	**GFLOPS/GOPS**
58											
59	**exclude C read/write each step, register bypass approach, stride optimized VR loads**										
60	**MACs/cycle**	**64,0**	**32,0**	**5,3**	**341,3**	**170,7**	**42,7**	**2730,7**	**1365,3**	**252,1**	**MACs/cycle**
61	**Mem BW for A,B load**	**16**	**16**	**16**	**43**	**43**	**64**	**64**	**64**	**63**	**Bytes/cycle**
62	**Performance**	**256**	**128**	**21**	**2048**	**1024**	**256**	**10923**	**5461**	**1008**	**GFLOPS/GOPS**

INNER PRODUCT: LOW-END „EMBEDDED“ CORE





mtype	rows	cols	RMUL	CMUL
C	1	8	1	4
A	1	2	1	1
B	2	8	1	8

fp64 += fp64*fp64

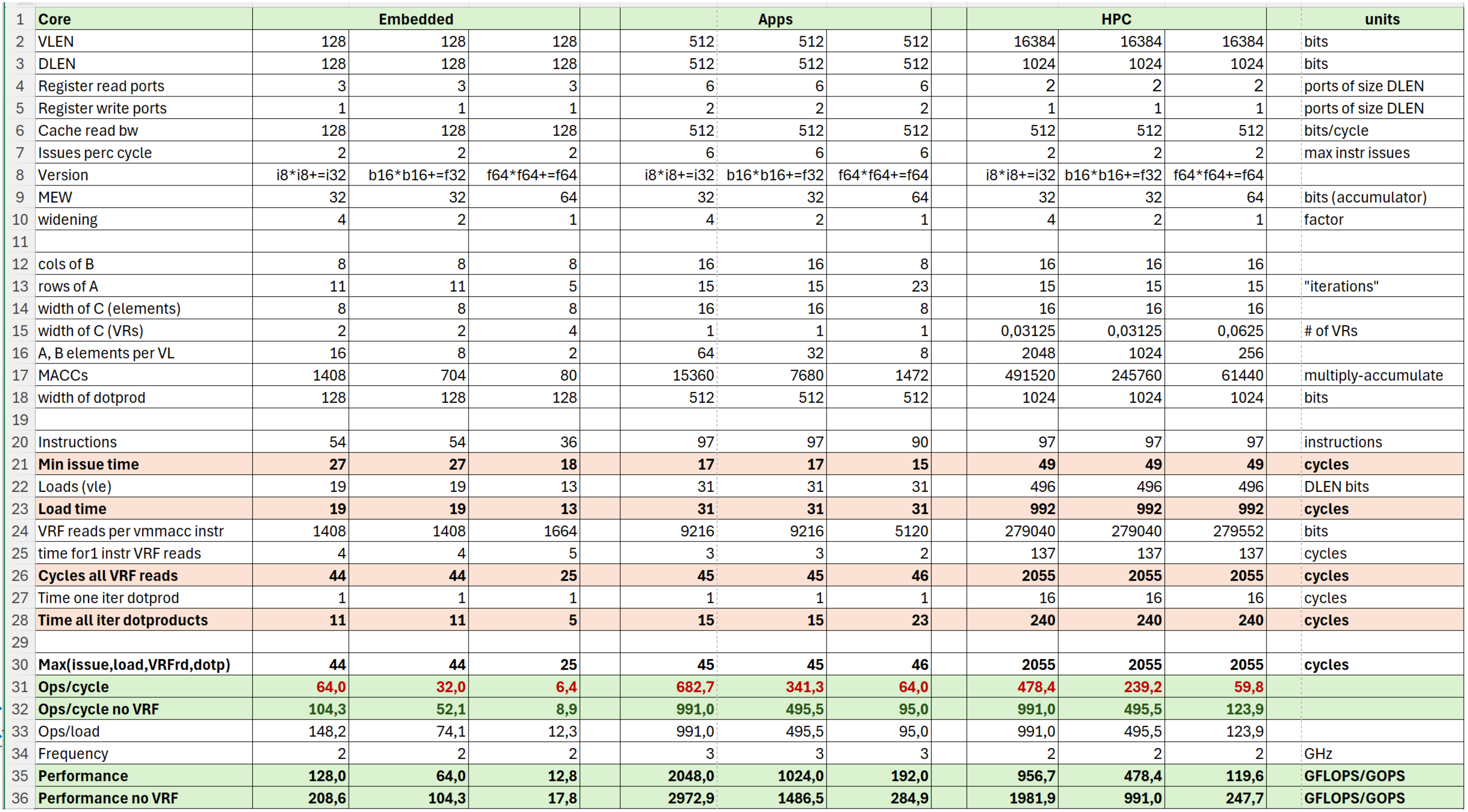
mtype	rows	cols	RMUL	CMUL
C	1	8	1	2
A	1	8	1	1
B	8	8	1	8

fp32 += bf16*bf16

mtype	rows	cols	RMUL	CMUL
C	1	8	1	2
A	1	16	1	1
B	16	8	1	8

int32 += int8*int8

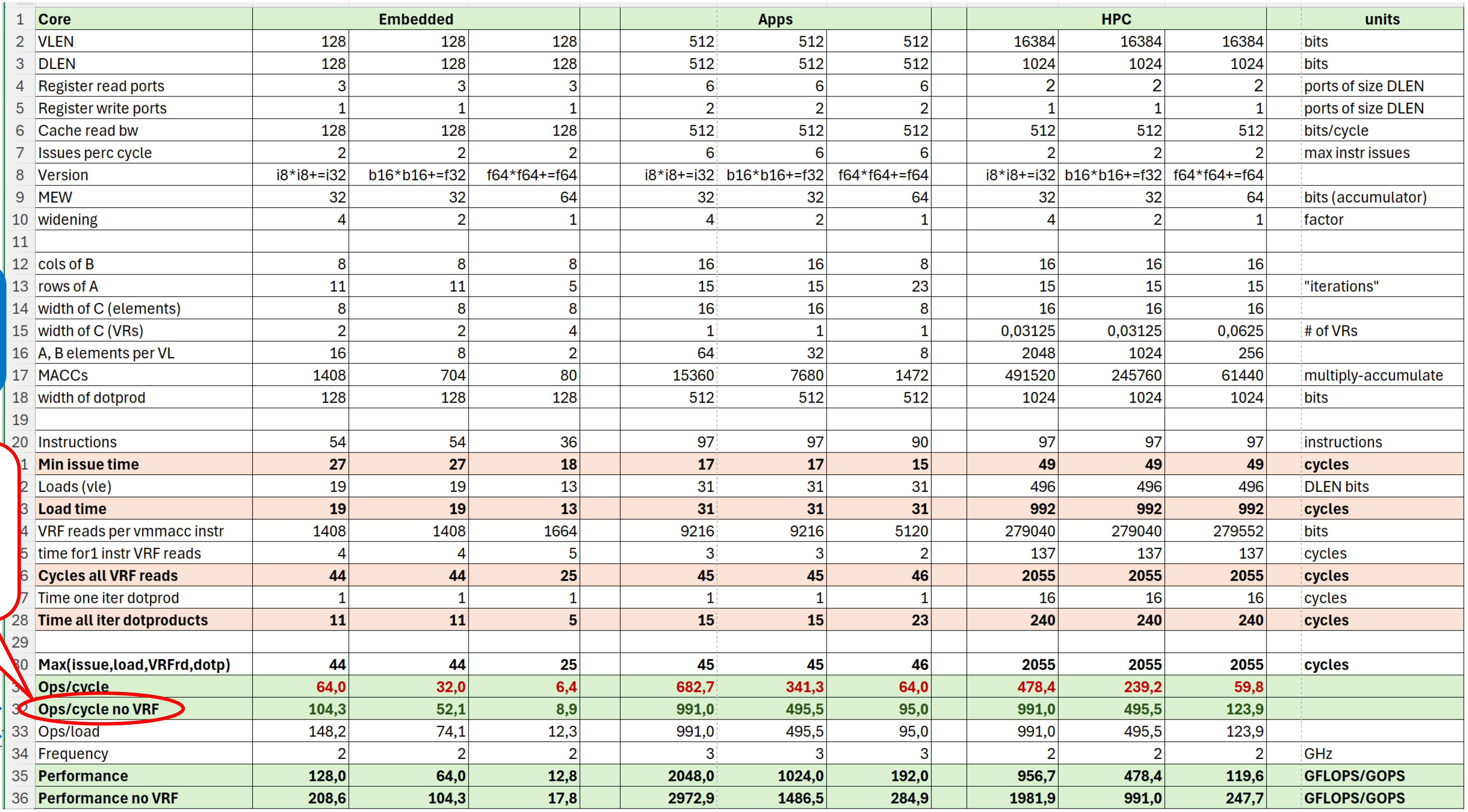
Evaluation – Inner Product



Core	Embedded			Apps			HPC			units
VLEN	128	128	128	512	512	512	16384	16384	16384	bits
DLEN	128	128	128	512	512	512	1024	1024	1024	bits
Register read ports	3	3	3	6	6	6	2	2	2	ports of size DLEN
Register write ports	1	1	1	2	2	2	1	1	1	ports of size DLEN
Cache read bw	128	128	128	512	512	512	512	512	512	bits/cycle
Issues perc cycle	2	2	2	6	6	6	2	2	2	max instr issues
Version	i8*i8+=i32	b16*b16+=f32	f64*f64+=f64	i8*i8+=i32	b16*b16+=f32	f64*f64+=f64	i8*i8+=i32	b16*b16+=f32	f64*f64+=f64	
MEW	32	32	64	32	32	64	32	32	64	bits (accumulator)
widening	4	2	1	4	2	1	4	2	1	factor
cols of B	8	8	8	16	16	8	16	16	16	
rows of A	11	11	5	15	15	23	15	15	15	"iterations"
width of C (elements)	8	8	8	16	16	8	16	16	16	
width of C (VRs)	2	2	4	1	1	1	0,03125	0,03125	0,0625	# of VRs
A, B elements per VL	16	8	2	64	32	8	2048	1024	256	
MACCs	1408	704	80	15360	7680	1472	491520	245760	61440	multiply-accumulate
width of dotprod	128	128	128	512	512	512	1024	1024	1024	bits
Instructions	54	54	36	97	97	90	97	97	97	instructions
Min issue time	**27**	**27**	**18**	**17**	**17**	**15**	**49**	**49**	**49**	**cycles**
Loads (vle)	19	19	13	31	31	31	496	496	496	DLEN bits
Load time	**19**	**19**	**13**	**31**	**31**	**31**	**992**	**992**	**992**	**cycles**
VRF reads per vmmacc instr	1408	1408	1664	9216	9216	5120	279040	279040	279552	bits
time for1 instr VRF reads	4	4	5	3	3	2	137	137	137	cycles
Cycles all VRF reads	**44**	**44**	**25**	**45**	**45**	**46**	**2055**	**2055**	**2055**	**cycles**
Time one iter dotprod	1	1	1	1	1	1	16	16	16	cycles
Time all iter dotproducts	**11**	**11**	**5**	**15**	**15**	**23**	**240**	**240**	**240**	**cycles**
Max(issue,load,VRFrd,dotp)	**44**	**44**	**25**	**45**	**45**	**46**	**2055**	**2055**	**2055**	**cycles**
Ops/cycle	**64,0**	**32,0**	**6,4**	**682,7**	**341,3**	**64,0**	**478,4**	**239,2**	**59,8**	
Ops/cycle no VRF	**104,3**	**52,1**	**8,9**	**991,0**	**495,5**	**95,0**	**991,0**	**495,5**	**123,9**	
Ops/load	148,2	74,1	12,3	991,0	495,5	95,0	991,0	495,5	123,9	
Frequency	2	2	2	3	3	3	2	2	2	GHz
Performance	**128,0**	**64,0**	**12,8**	**2048,0**	**1024,0**	**192,0**	**956,7**	**478,4**	**119,6**	**GFLOPS/GOPS**
Performance no VRF	**208,6**	**104,3**	**17,8**	**2972,9**	**1486,5**	**284,9**	**1981,9**	**991,0**	**247,7**	**GFLOPS/GOPS**

Match ~ Option F evaluation

EVALUATION – INNER PRODUCT



Core	Embedded			Apps			HPC			units
VLEN	128	128	128	512	512	512	16384	16384	16384	bits
DLEN	128	128	128	512	512	512	1024	1024	1024	bits
Register read ports	3	3	3	6	6	6	2	2	2	ports of size DLEN
Register write ports	1	1	1	2	2	2	1	1	1	ports of size DLEN
Cache read bw	128	128	128	512	512	512	512	512	512	bits/cycle
Issues perc cycle	2	2	2	6	6	6	2	2	2	max instr issues
Version	i8*i8+=i32	b16*b16+=f32	f64*f64+=f64	i8*i8+=i32	b16*b16+=f32	f64*f64+=f64	i8*i8+=i32	b16*b16+=f32	f64*f64+=f64	
MEW	32	32	64	32	32	64	32	32	64	bits (accumulator)
widening	4	2	1	4	2	1	4	2	1	factor
cols of B	8	8	8	16	16	8	16	16	16	
rows of A	11	11	5	15	15	23	15	15	15	"iterations"
width of C (elements)	8	8	8	16	16	8	16	16	16	
width of C (VRs)	2	2	4	1	1	1	0,03125	0,03125	0,0625	# of VRs
A, B elements per VL	16	8	2	64	32	8	2048	1024	256	
MACCs	1408	704	80	15360	7680	1472	491520	245760	61440	multiply-accumulate
width of dotprod	128	128	128	512	512	512	1024	1024	1024	bits
Instructions	54	54	36	97	97	90	97	97	97	instructions
Min issue time	**27**	**27**	**18**	**17**	**17**	**15**	**49**	**49**	**49**	**cycles**
Loads (vle)	19	19	13	31	31	31	496	496	496	DLEN bits
Load time	**19**	**19**	**13**	**31**	**31**	**31**	**992**	**992**	**992**	**cycles**
VRF reads per vmmacc instr	1408	1408	1664	9216	9216	5120	279040	279040	279552	bits
time for1 instr VRF reads	4	4	5	3	3	2	137	137	137	cycles
Cycles all VRF reads	**44**	**44**	**25**	**45**	**45**	**46**	**2055**	**2055**	**2055**	**cycles**
Time one iter dotprod	1	1	1	1	1	1	16	16	16	cycles
Time all iter dotproducts	**11**	**11**	**5**	**15**	**15**	**23**	**240**	**240**	**240**	**cycles**
Max(issue,load,VRFrd,dotp)	**44**	**44**	**25**	**45**	**45**	**46**	**2055**	**2055**	**2055**	**cycles**
Ops/cycle	**64,0**	**32,0**	**6,4**	**682,7**	**341,3**	**64,0**	**478,4**	**239,2**	**59,8**	
Ops/cycle no VRF	**104,3**	**52,1**	**8,9**	**991,0**	**495,5**	**95,0**	**991,0**	**495,5**	**123,9**	
Ops/load	148,2	74,1	12,3	991,0	495,5	95,0	991,0	495,5	123,9	
Frequency	2	2	2	3	3	3	2	2	2	GHz
Performance	**128,0**	**64,0**	**12,8**	**2048,0**	**1024,0**	**192,0**	**956,7**	**478,4**	**119,6**	**GFLOPS/GOPS**
Performance no VRF	**208,6**	**104,3**	**17,8**	**2972,9**	**1486,5**	**284,9**	**1981,9**	**991,0**	**247,7**	**GFLOPS/GOPS**

Clarified by Jose during the meeting: „F" uses banking of VRF (2x256b).

Implies that B VRs are not transferred into the fat dotprod unit each time they are used.

Match ~ Option F evaluation

C = ALPHA · A · B + BETA · C

Matrix data types: A,B,C: FP32

```
vmts X(tc),X(M|N<<16),e32,mtc,rmo
vmts X(ta),X(M|K<<16),e32,mta,rmo
vmts X(tb),X(K|N<<16),e32,mtb,rmo
IF (RMUL(tc)*CMUL(tc)==1 &&
    RMUL(ta)*CMUL(ta)==1 &&
    RMUL(ta)*CMUL(ta)==1) THEN        # 16 steps
    call gemm_16_steps_1_1_1
ELSE IF (RMUL(tc)*CMUL(tc)==16) THEN  # one step
    IF (RMUL(ta)*CMUL(ta)==4) THEN
        IF (RMUL(tb)*CMUL(tb)==4) THEN
            call gemm_one_step_16_4_4
        ELSE IF (RMUL(tb)*CMUL(tb)==8) THEN
            call gemm_one_step_16_4_8
        ELSE
            ...
        ENDIF
ENDIF
```

```
gemm_one_step_16_4_4:

 tm = ROWS(tc); tn = COLS(tc); tk = ROWS(tb)
 for m = 0, M, m += tm do
    for n = 0, N, n += tn do
        vsetvli zero,x(-1),e32,m8,ta,ma
        vfmv.v.f v16,f(0.0f)
        vfmv.v.f v24,f(0.0f)
        vmmset(tc,ta,tb)
        for k = 0, K, k += tk do
            vmlt v8,x(&A[m,k]),x(rstr_A|cstr_A<<31|mta<<62)
            vmlt v12,x(&B[k,n]),x(rstr_B|cstr_B<<31|mtb<<62)
            vfmmacc v16,v8,v12
        vmlt v0,x(&C[m,n]),x(rstr_C|cstr_C<<31|mtc<<62)
        vsetvli zero,X(-1),e32,m8,ta,ma
        vfmul.vf v16,v16,alpha
        vfmul.vf v24,v24,alpha
        vfmacc.vf v16,v0,beta
        vfmacc.vf v24,v8,beta
        vmst v16,x(&C[m,n]),x(rstr_C|cstr_C<<31|mtc<<62)
```

SUMMARY

✓ISA can be portable

- **Need flexible query of hardware capabilities**
- **Tile load/store commands**

✓Implementor has the choice

- Choose hardware (sequencer) vs. software balance
- Inner or outer product
- Optimizations: buffering B or Accumulator or ...

✓Can map most of the approaches

- Segments of A, C tiles not yet handled
- (Software should know if approach is weight stationary or output stationary)

openchip